References

Collaborated with Crystal Huang. See Exponentiation by squaring – Wikipedia.

Question 1: InsertionSort *vs.* QuickSort

Recall that in the worst case the running time of (non-randomized) QuickSort, that uses the last element as the pivot, and InsertionSort are $\Theta(n^2)$. Refer to the lecture notes for the QuickSort implementation.

1. Give an example of an array of length n where both take time $\Omega(n^2)$. Justify your answer.

Solution: Let $A[1, \ldots, n]$ be an n-element array where $A[i] > A[i+1]$ for all $1 \leq i < n$.

Lemma I.

Claim: QuickSort(A) has running time $\Omega(n^2)$.

We will analyze each recursive call i to QuickSort, where $i \geq 1$.

- Consider $i = 1$. This recursive call operates on A:

$$A = (A[1], A[2], \ldots, A[n]).$$

  Since the pivot is chosen to be the last element of A, and since A is sorted in reverse order, the last element is the least element in A. In other words, the pivot is chosen to be $A[n]$ where $A[n] \leq A[i]$ for all $1 < i < n$.

  During the Partition step, all elements are greater than or equal to $A[n]$, so no swaps are performed. Finally, $A[1]$ and $A[n]$ are swapped to place the pivot in its correct position. Note the resulting array A_1:

$$A_1 = (A[n], A[2], A[3] \ldots, A[n-1], A[1]).$$

- Consider $i = 2$. This recursive call operates on the subarray of A_1:

$$(A[2], A[3], \ldots, A[n-1], A[1]).$$

  The pivot is chosen to be the last element of this subarray, which is the value $A[1]$. Because of the previous swap, in this case $A[1]$ is the largest element in the subarray.

  During the Partition step, all elements in the subarray are less than or equal to $A[1]$, so no swaps are performed. Finally, the values $A[n-1]$ and $A[1]$ are swapped to place the pivot in its correct position. Note the resulting array A_2:

$$A_2 = (A[2], A[3], \ldots, A[1], A[n-1]).$$

For each odd recursive step $i = 2k + 1$, the pivot is the least element; for each even recursive step $i = 2k$, the pivot is the greatest element. Thus, for each recursive step, the pivot is an

extreme within its working subarray. As a result, the PARTITION step results in a lopsided set of subarrays, with one of length $n-1$ and the other of length 0.

On every recursive step, the size of the current working subarray is reduced by exactly 1. Therefore, there are n levels, and on each level i, there are $n-i$ operations performed. Thus the amount of work is no less than

$$\sum_{i=1}^{n}(n-i) = n + (n-1) + (n-2) + \cdots = \Omega(n).$$

Therefore QUICKSORT(A) has running time $\Omega(n^2)$.

Lemma II.

Claim. INSERTIONSORT(A) has running time $\Omega(n^2)$.

We will analyze each iteration i of INSERTIONSORT, where $i \geq 1$.

- Consider $i = 1$. This iteration operates on $A[2]$. Note $A[2] < A[1]$, so a comparison and a shift is required.
- Consider $i = 2$. This iteration operates on $A[3]$. Note $A[3] < A[2]$ and $A[3] < A[1]$, so two comparisons and two shifts are required.

For each iteration i, there are i-many comparisons and shifts required to insert $A[i+1]$ into its sorted position. Thus the amount of work is no less than

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \Omega(n^2).$$

Therefore INSERTIONSORT(A) has running time $\Omega(n^2)$.

Proof.

From Lemma I, We have that QUICKSORT(A) has running time $\Omega(n^2)$.

From Lemma II, We have that INSERTIONSORT(A) has running time $\Omega(n^2)$.

Hence proven. □

2. Give an example of an array of length n where (non-randomized) QUICKSORT runs in $\Omega(n^2)$ but INSERTION-SORT runs in time $O(n)$. Justify your answer.

Solution: Let $A[1, \ldots, n]$ be an n-element array where $A[i] < A[i+1]$ for all $1 \leq i < n$.

Lemma I.

Claim: QUICKSORT(A) has running time $\Omega(n^2)$.

We will analyze each recursive call i to QUICKSORT, where $i \geq 1$.

- Consider $i = 1$. This recursive call operates on A:

  $$A = (A[1], A[2], \ldots, A[n]).$$

  Since the pivot is chosen to be the last element of A, and since A is sorted, the last element is the greatest element in A. In other words, the pivot is chosen to be $A[n]$ where $A[n] \geq A[i]$ for all $1 < i < n$.

  During the PARTITION step, all elements are less than or equal to $A[n]$, so no swaps are performed. The pivot $A[n]$ remains in its correct position. There is no change to the array.

- Consider $i = 2$. This recursive call operates on the subarray of A:

  $$(A[1], A[2], \ldots, A[n-1]).$$

  Again, the pivot is chosen to be the last element of this subarray, which is the value $A[n-1]$, the largest element in this subarray of A.

  During the PARTITION step, all elements in the subarray are less than or equal to $A[n-1]$, so no swaps are performed. Again, the pivot $A[n-1]$ remains in its correct position. Again, there is no change to the array.

For each recursive step i, the pivot is the greatest element. Thus, for each recursive step, the pivot is the maximum within its working subarray. As a result, the PARTITION step results in a lopsided set of subarrays, with one of length $n-1$ and the other of length 0.

On every recursive step, the size of the current working subarray is reduced by exactly 1. Therefore, there are n levels, and on each level i, there are $n - i$ operations performed. Thus the amount of work is no less than

$$\sum_{i=1}^{n}(n-i) = n + (n-1) + (n-2) + \cdots = \Omega(n).$$

Therefore QUICKSORT(A) has running time $\Omega(n^2)$.

Lemma II.

Claim. INSERTIONSORT(A) has running time $\Omega(n^2)$.

We will analyze each iteration i of INSERTIONSORT, where $i \geq 1$.

- Consider $i = 1$. This iteration operates on $A[2]$. Note $A[2] > A[1]$, so only a comparison is required.
- Consider $i = 2$. This iteration operates on $A[3]$. Note $A[3] > A[2]$, so only a comparison is required.

For each iteration i, there is 1 comparison required to ensure $A[i + 1]$ is in its sorted position. Let c be the positive constant that denotes the running time of each comparison operation. Thus the amount of work is no less than

$$\sum_{i=1}^{n} c = cn = \Omega(n).$$

Therefore InsertionSort(A) has running time $\Omega(n)$.

Proof.

From Lemma I, We have that QuickSort(A) has running time $\Omega(n^2)$.

From Lemma II, We have that InsertionSort(A) has running time $\Omega(n)$.

Hence proven. □

Question 2: Fast exponentiation

In this problem, you will design an algorithm to compute 2024^n given $n \in \mathbb{N}$ as input. In each case, prove the correctness of your algorithm, and an upper bound on the number of multiplications used.

1. Using $n - 1$ many multiplications.

Solution: NaiveExponentiation(n) where $n \in \mathbb{N}$:

- if $n = 1$, then return 2024;
- otherwise, return the product $2024 \times$ NaiveExponentiation($n - 1$).

Lemma I.

Claim. NaiveExponentiation(n) $= 2024^n$ for all $n \in \mathbb{N}$. We can demonstrate the claim by induction on n.

Basis. Consider $n = 1$. We have NaiveExponentiation(1) $= 2024 = 2024^1$. The claim holds in the base case.

Hypothesis. Consider $n = k$ where $k > 1$. Assume that NaiveExponentiation(k) $= 2024^k$.

Inductive step. Consider $n = k + 1$. Then

NaiveExponentiation($k + 1$) $= 2024 \times$ NaiveExponentiation($(k + 1) - 1$). That is,

NaiveExponentiation($k + 1$) $= 2024 \times$ NaiveExponentiation(k).

By the inductive hypothesis, we have NaiveExponentiation(k) $= 2024^k$. Thus

NaiveExponentiation($k + 1$) $= 2024 \times 2024^k$. Ergo,

NaiveExponentiation($k + 1$) $= 2024^{k+1}$, thus completing the inductive step.

Hence, by the principle of mathematical induction, the NaiveExponentiation algorithm produces the correct result for all $n \in \mathbb{N}$.

Lemma II.

Claim. NaiveExponentiation(n) uses at most $n - 1$ multiplications for all $n \in \mathbb{N}$. We can demonstrate the claim by induction on n.

Basis. Consider $n = 1$. Since $n = 1$, the number of multiplications is $0 = 1 - 1 \leq n - 1$. The claim holds in the base case.

Hypothesis. Consider $n = k$ where $k > 1$. Assume that NaiveExponentiation(k) uses at most $k - 1$ multiplications.

Inductive step. Consider $n = k + 1$. Then the number of multiplications is 1, plus the number of multiplications used by NaiveExponentiation($(k + 1) - 1$). That is, 1 plus the number of multiplications used by NaiveExponentiation(k). By the inductive hypothesis, the number of multiplications used by NaiveExponentiation(k) is at most $k - 1$. Therefore, the number of multiplications used by NaiveExponentiation($k + 1$) is at most $1 + (k - 1) = (k + 1) - 1$, thus completing the inductive step.

Hence, by the principle of mathematical induction, the NaiveExponentiation algorithm uses at most $n - 1$ multiplications for all $n \in \mathbb{N}$.

Proof.

By Lemma I, NaiveExponentiation(n) computes 2024^n for all $n \in \mathbb{N}$.

By Lemma II, NaiveExponentiation(n) uses at most $n - 1$ multiplications for all $n \in \mathbb{N}$.

Ergo, for all $n \in \mathbb{N}$, we have demonstrated that NaiveExponentiation(n) computes 2024^n using at most $n - 1$ multiplications. □

2. Using $O(\log_2 n)$ many multiplications, assuming n is a power of 2, i.e., $n = 2^k$.

Solution: SquareExponentiation(n) where $n = 2^x$ for $x \geq 0$:

- if $n = 2^0$, then return 2024;
- otherwise, compute $y \leftarrow$ SquareExponentiation($n/2$), and return $y \times y$.

Lemma I.

Claim. SquareExponentiation(n) $= 2024^n$ for $n = 2^x$ where $x \geq 0$. We can demonstrate the claim by induction on n.

Basis. Consider $n = 2^0$. We have SquareExponentiation(2^0) $= 2024 = 2024^1$. The claim holds in the base case.

Hypothesis. Consider $n = 2^k$ where $k > 0$. Assume that SquareExponentiation(2^k) $= 2024^{2^k}$.

Inductive step. Consider $n = 2^{k+1}$. Then

$y =$ SquareExponentiation($2^{k+1}/2$) $=$ SquareExponentiation(2^k). By the inductive hypothesis, we have SquareExponentiation(2^k) $= 2024^{2^k}$. Thus $y = 2024^{2^k}$. Now

SquareExponentiation(2^{k+1}) $= y \times y$, so SquareExponentiation(2^{k+1}) $= 2024^{2^k} \times 2024^{2^k} = 2024^{2^k+2^k} = 2024^{2 \cdot 2^k} = 2024^{2^{k+1}}$, thus completing the inductive step.

Hence, by the principle of mathematical induction, the SquareExponentiation algorithm produces the correct result for all $n = 2^x$ where $x \geq 0$.

Lemma II.

Claim. SquareExponentiation(n) uses $O(\log_2 n)$ multiplications for $n = 2^x$ where $x \geq 0$. Let $M(n)$ denote the number of multiplications used by Square Exponentiation(n). We claim that $M(n) = O(\log_2 n)$. We can demonstrate the claim by induction on n.

Basis. Consider $n = 2^0$. Then there are no multiplications, so $M(2^0) = 0$. Since $\log_2 n \geq 0$ for $n \geq 1$, we can choose any positive constant C and any $n_0 \geq 1$ such that $0 \leq C \log_2 n$ for $n > n_0$. Therefore $M(n) = O(\log_2 n)$ holds in the base case.

Hypothesis. Consider $n = 2^k$ where $k > 0$. Assume that $M(2^k) = O(\log_2 k)$.

Inductive step. Consider $n = 2^{k+1}$. Then the number of multiplications is 1 (to compute $y \times y$), plus the number of multiplications used by the recursive call. That is,

$$\begin{aligned} M(2^{k+1}) &= 1 + M\left(\frac{2^{k+1}}{2}\right) \\ &= 1 + M(2^k). \end{aligned}$$

By the inductive hypothesis, we have $M(2^k) = O(\log_2(k+1))$. Asymptotically,

$$\begin{aligned} M(2^{k+1}) &= 1 + O(\log_2(k+1)) \\ &= O(1) + O(\log_2 k) \\ &= O(\log_2 k). \end{aligned}$$

This completes the inductive step.

Hence, by the principle of mathematical induction, the SQUAREEXPONENTIATION algorithm uses $O(\log_2 n)$ many multiplications for all $n = 2^x$ where $x \geq 0$.

Proof.

By Lemma I, SQUAREEXPONENTIATION(n) computes 2024^n for all $n = 2^x$ where $x \geq 0$.

By Lemma II, SQUAREEXPONENTIATION(n) uses at $O(\log_2 n)$ many multiplications for all $n = 2^x$ where $x \geq 0$.

Ergo, for all $n = 2^x$ where $x \geq 0$, we have demonstrated that SQUAREEXPONENTIATION(n) computes 2024^n using $O(\log_2 n)$ many multiplications. □

3. Using $O(\log_2 n)$ many multiplications for *any* n (not necessarily a power of 2).

Solution:

Algorithm I. FastExponentiation(b, n) where $n \in \mathbb{N}$:

- if $n = 1$, then return b;
- otherwise, compute $b' \leftarrow b \times b$, and:
  - if n is even, then return FastExponentiation$(b', n/2)$;
  - otherwise, return the product $b \times$ FastExponentiation$(b', (n-1)/2)$.

Algorithm II. FastPow2024(n) where $n \in \mathbb{N}$: return the result FastExponentiation$(2024, n)$.

Lemma I.

Claim. FastExponentiation$(b, n) = b^n$ for all $b, n \in \mathbb{N}$. We can demonstrate the claim by induction on n.

Basis. Consider $n = 1$. We have FastExponentiation$(b, 1) = b = b^1$ for all $b \in \mathbb{N}$. The claim holds in the base case.

Hypothesis. Consider $1 < n \leq k$. Assume that FastExponentiation$(b, k) = b^k$ for all $b \in \mathbb{N}$.

Inductive step. Consider $n = k + 1$. Since $k + 1 > 1$, we compute $b' = b \times b$. Of course, since $b \in \mathbb{N}$, we have $b' \in \mathbb{N}$. Since $k \in \mathbb{N}$, either $k + 1$ is even or $k + 1$ is odd:

- Suppose $k + 1$ is even. Now,
  FastExponentiation$(b, k + 1) =$ FastExponentiation$(b', \frac{k+1}{2})$.
  Since $k+1 \in \mathbb{N}$ and $k+1$ is even, we have $\frac{k+1}{2} \in \mathbb{N}$. By the strong induction hypothesis, FastExponentiation$(b', \frac{k+1}{2}) = {b'}^{\frac{k+1}{2}} = (b \times b)^{\frac{k+1}{2}} = (b^2)^{\frac{k+1}{2}} = b^{k+1}$. Ergo
  FastExponentiation$(b, k + 1) = b^{k+1}$.
- Suppose instead $k + 1$ is odd. Now,
  FastExponentiation$(b, k + 1) = b \times$ FastExponentiation$(b', \frac{(k+1)-1}{2})$.
  Since $k+1 \in \mathbb{N}$ and $k+1$ is odd, we have $\frac{(k+1)-1}{2} \in \mathbb{N}$. By the strong induction hypothesis, FastExponentiation$(b', \frac{(k+1)-1}{2}) = {b'}^{\frac{(k+1)-1}{2}} = {b'}^{\frac{k}{2}} = (b \times b)^{\frac{k}{2}} = (b^2)^{\frac{k}{2}} = b^k$. Ergo
  FastExponentiation$(b, k + 1) = b \times b^k = b^{k+1}$.

In all cases, FastExponentiation$(b, k + 1) = b^{k+1}$ for all $b \in \mathbb{N}$. This completes the inductive step.

Hence, by the principle of mathematical induction, the FastExponentiation algorithm produces the correct result for all $b, n \in \mathbb{N}$.

Lemma II.

Claim. FastExponentiation$(b, n) = b^n$ uses $O(\log_2 n)$ multiplications for all $b, n \in \mathbb{N}$. Let $M(b, n)$ denote the number of multiplications used by FastExponentiation(b, n). We claim that $M(b, n) = O(\log_2 n)$. We can demonstrate the claim by induction on n.

Basis. Consider $n = 1$. Then, regardless of the base b, there are no multiplications, so $M(b, 1) = 0$ for all $b \in \mathbb{N}$. Since $\log_2 n \geq 0$ for $n \geq 1$, we can choose any positive constant C and any $n_0 \geq 1$ such that $0 \leq C \log_2 n$ for $n \geq n_0$. Therefore $M(b, n) = O(\log_2 n)$ holds for all $b \in \mathbb{N}$ in the base case.

Hypothesis. Consider $1 < n \leq k$. Assume that $M(b, k) = O(\log_2 n)$ for all $b \in \mathbb{N}$.

Inductive step. Consider $n = k + 1$. Since $k + 1 > 1$, we know 1 multiplication is required to compute $b' = b \times b$. Since $k \in \mathbb{N}$, either $k + 1$ is even or $k + 1$ is odd:

- Suppose $k + 1$ is even. The number of additional multiplications is $M\left(b', \frac{k+1}{2}\right)$. By the strong induction hypothesis, we have $M\left(b', \frac{k+1}{2}\right) = O(\log_2 n)$.
- Suppose instead $k+1$ is odd. The number of additional multiplications is $M\left(b', \frac{(k+1)-1}{2}\right)$. By the strong induction hypothesis, we have $M\left(b', \frac{(k+1)-1}{2}\right) = M\left(b', \frac{k}{2}\right) = O(\log_2 n)$.

In all cases, the number of additional multiplications is $O(\log_2 n)$. Considering also the single multiplication used to compute $b' = b \times b$, asymptotically we have

$$\begin{aligned} M(b, k + 1) &= 1 + O(\log_2 n) \\ &= O(1) + O(\log_2 n) \\ &= O(\log_2 n). \end{aligned}$$

This completes the inductive step.

Hence, by the principle of mathematical induction, the FastExponentiation algorithm uses $O(\log_2 n)$ many multiplications for all $b, n \in \mathbb{N}$.

Proof.

By construction, the FastPow2024(n) algorithm returns FastExponentiation$(2024, n)$ and uses exactly as many multiplications as the FastExponentiation$(2024, n)$ algorithm.

By Lemma I, FastExponentiation$(2024, n)$ computes 2024^n for all $n \in \mathbb{N}$. Therefore, the FastPow2024 algorithm computes 2024^n for all $n \in \mathbb{N}$.

By Lemma II, FastExponentiation$(2024, n)$ uses $O(\log_2 n)$ many multiplications for all $n \in \mathbb{N}$. Therefore, the FastPow2024 algorithm uses $O(\log_2 n)$ many multiplications for all $n \in \mathbb{N}$.

Ergo, for all $n \in \mathbb{N}$, we have demonstrated that FastPow2024(n) computes 2024^n using $O(\log_2 n)$ many multiplications. □

Question 3: Close to sorted

Suppose you are given a list A of n distinct numbers. You are guaranteed that this list is 'close to sorted' in the following sense: if A_{sorted} denotes the list fully sorted in increasing order, then A_{sorted} differs from A in at most $\log_2 n$ many positions. Intuitively, this says that at most $\log_2 n$ elements are out of place in the original list. For instance, in the list

$$(2, 5, 9, 11, 20, 14, 15, 12, 25, 30)$$

only two elements are out of place (20 and 12), since after sorting, we get

$$(2, 5, 9, 11, 12, 14, 15, 20, 25, 30) \ .$$

The following exercises will ultimately lead to an algorithm which sorts A in time $O(n)$ [1]. Answer each exercise.

1. Prove that the leftmost (first) out of place element must be too big (not too small) for its place.

Solution: *Claim.* Let k represent the position of the leftmost out-of-place element in A. Then $A[k]$ is too big (not too small) for its position.

Proof. Assume, for the sake of contradiction, that k is too small for its position. Since $A[k]$ is too small for its position, we know $k > 1$. Otherwise, if $k = 1$, it could not be too small for its position. Of course, since $A[k]$ is too small for its position, $A[k] < A[k-1]$.

This implies that $A[k-1] > A[k]$, meaning that $A[k-1]$ is too big for its place. $A[k-1]$ is out of place and $k - 1 < k$. Therefore, $A[k-1]$ is out of place and *further left* than $A[k]$. This contradicts the hypothesis that $A[k]$ is the leftmost out-of-place element in A.

We conclude that the leftmost out-of-place element in A must *not* be too small for its place. Hence, the leftmost out-of-place element in A must be too big for its place. □

[1] Note that sorting algorithms take $O(n \log n)$ time without any assumptions on the inputs. Here, we are able to get the faster $O(n)$ run-time by *assuming* that the input array A is already close to being sorted.

2. Suppose we construct a stack S as follows: We go through A from left to right, pushing elements one by one into S as long as the next element from A is larger than the top element s of the stack so far. If at any step i the element $A[i]$ which we are currently considering in A is smaller than s, we instead pop s from S and continue. This idea is described in the pseudo code below:

Algorithm 1 Construct increasing subsequence S

Require: A and an empty stack S.
Ensure: S representing an increasing subsequence of A

```
n ← len(A)
for i = 1 to n do
  if S is empty or S.topElement() ≤ A[i] then
    S.push(A[i])
  else
    S.pop()
  end if
end for

return S
```

If A is initially $(2, 5, 9, 11, 20, 14, 15, 12, 25, 30)$, what will S be after Algorithm 1 has run?

Solution: Let $A[1, \ldots, n] = (2, 5, 9, 11, 20, 14, 15, 12, 25, 30)$. Let S be an empty stack.

Let S_i denote S before iteration $1 \leq i \leq n + 1$. For example, S_1 represents the stack before iteration 1 and S_2 represents the stack after iteration 1 but before iteration 2. S_{n+1} represents S after the final iteration. Stepping through the algorithm:

$$
\begin{aligned}
S_1 &= () && \textit{before the first iteration.} \\
S_2 &= (2) && \textit{since } S_1 \textit{ is empty.} \\
S_3 &= (2, 5) && \textit{since } 2 \leq 5. \\
&\vdots && \vdots \\
S_6 &= (2, 5, 9, 11, 20) && \textit{since } 5 \leq 9 \leq 11 \leq 20. \\
S_7 &= (2, 5, 9, 11) && \textit{since } S \textit{ is not empty, } 20 \nleq 14. \\
S_8 &= (2, 5, 9, 11, 15) && \textit{since } 11 \leq 15. \\
S_9 &= (2, 5, 9, 11) && \textit{since } S \textit{ is not empty, } 15 \nleq 12. \\
&\vdots && \vdots \\
S_{11} &= (2, 5, 9, 11, 25, 30) && \textit{since } 11 \leq 25 \leq 30.
\end{aligned}
$$

After Algorithm 1 has run, $S = (2, 5, 9, 11, 25, 30)$.

3. Prove that S is an increasing subsequence of A (the elements of the output stack are increasing from bottom to top).

Solution: Invariant I.

Claim. Let $A[1, \ldots, n]$ be an array and S be an empty stack. Consider Algorithm 1.

Let S_i denote S before iteration $1 \leq i \leq n+1$. For example, S_1 represents the stack before iteration 1 and S_2 represents the stack after iteration 1 but before iteration 2.

We want to demonstrate that, for every iteration of the loop in Algorithm 1, the stack S is a monotonically increasing subsequence of A.

That is, for all $i \leq n+1$, for all $s \in S_i$, we have $s \in A$, and for all $1 \leq j \leq \text{len}(S_i)$, we have $S_i[j] \leq S_i[j+1]$.

We can prove this invariant by induction on i.

Basis. Consider $i = 1$ (before iteration 1). Here S_1 is an empty stack. Vacuously, for all $s \in S_1$, we have $s \in A$. Similarly, for all $1 \leq j < \text{len}(S_1)$, we have $S_1[j] \leq S_1[j+1]$. The invariant holds in the base case.

Hypothesis. Consider $1 < i \leq k \leq \text{len}(A)$ (before iteration k). Assume that for all $s \in S_k$, we have $s \in A$, and for all $1 \leq j \leq \text{len}(S_k)$, we have $S_k[j] \leq S_k[j+1]$.

Inductive step. Consider $i = k+1$ (after iteration k, but before iteration $k+1$):

- Suppose S_k was empty or $S_k[\text{len}(S_k)] \leq A[k]$. Note $S_k[\text{len}(S_k)]$ denotes the top element of the stack.

  Then the top element of the stack has been updated such that $S_{k+1}[\text{len}(S_{k+1})] = A[k]$.

  Of course $A[k] \in A$.

  Also, $A[k] \geq S_k[\text{len}(S_k)]$ so $S_{k+1}[\text{len}(S_{k+1}) - 1] \leq S_{k+1}[\text{len}(S_{k+1})]$.

  By the induction hypothesis, S_k is a monotonically increasing subsequence of A. By showing that the new top element in S_{k+1} is a member of A and is no smaller than the old top element, we have shown that S_{k+1} is also a monotonically increasing subsequence of A, and thus that the loop invariant holds.

- Suppose instead S_k was not empty and $S_k[\text{len}(S_k)] > A[k]$. Then the top element of the stack has been updated such that $S_{k+1}[\text{len}(S_{k+1})] = S_k[\text{len}(S_k) - 1]$. By the induction hypothesis, $S_k[\text{len}(S_k) - 1] \leq S_k[\text{len}(S_k)]$ and $S_k[\text{len}(S_k) - 1] \in A$. By substituting and leveraging the strong induction hypothesis, we know that $S_{k+1}[\text{len}(S_{k+1} - 1)] \leq S_{k+1}[\text{len}(S_{k+1})]$ and $S_{k+1}[\text{len}(S_{k+1})] \in A$.

  Essentially, we have shown that by removing the top element of S_k to produce S_{k+1}, we have returned to a prior state of S for which the strong induction hypothesis asserts that the invariant holds.

In all cases, the loop invariant holds for $i = k+1$.

Hence, by the principle of mathematical induction, for all $1 \leq i \leq n+1$, we can conclude that S_i is a monotonically increasing subsequence of A.

By proving the loop invariant at initialization ($i = 1$), maintenance ($1 \leq i \leq n$), and termination ($i = n+1$), we have shown that S is an increasing subsequence of A. $\square$

4. Prove that S contains all elements of A except at most $2\log_2 n$.

Solution: Invariant II.

Claim. Let $A[1, \ldots, n]$ be a close-to-sorted array such that at most $\log_2 n$ elements are out of place. Let S be an empty stack. Consider Algorithm 1.

We want to demonstrate that, for every iteration of the loop in Algorithm 1, the stack S contains all elements of A except at most $2\log_2 n$. In other words, denoting the number of excluded elements m, we want to show that $m \leq 2\log_2 n$.

Let S_i denote S before iteration $1 \leq i \leq n+1$. For example, S_1 represents the stack before iteration 1 and S_2 represents the stack after iteration 1 but before iteration 2. Similarly, let m_i denote the number of excluded elements of A before iteration i.

We can prove this invariant by induction on i.

Basis. Consider $i = 1$ (before iteration 1). Here S_1 is an empty stack and $m_1 = 0$ since no elements have been examined or excluded. Of course, for all $n \in \mathbb{N}$, we have $0 \leq 2\log_2 n$, so $m_1 \leq 2\log_2 n$. The loop invariant holds in the base cases.

Hypothesis. Consider $1 < i < k \leq n$ (before iteration k). Assume that $m_k \leq 2\log_2 n$.

Inductive step. Consider $i = k+1$ (after iteration k, but before iteration $k+1$):

- Suppose S_k was empty or $S_k[\text{len}(S_k)] \leq A[k]$. Note $S_k[\text{len}(S_k)]$ denotes the top element of the stack. No additional elements are excluded, so $m_{k+1} = m_k$. By the induction hypothesis, the loop invariant holds.

- Suppose instead S_k was not empty and $S_k[\text{len}(S_k)] > A[k]$. Then $A[k]$ is not pushed onto S_{k+1}, so one element is excluded. An element is popped from S_k, so another element is excluded. Considering these exclusions, $m_{k+1} = m_k + 2$.

  This case only occurs when $A[k]$ is out of place. Now, $A[k]$ is out of place for at most $\log_2 n$ many iterations (since A is almost sorted), so this case occurs at most $\log_2 n$ many times. On each such iteration, exactly 2 elements are excluded. Considering these bounds, taken together with the induction hypothesis ($m_k \leq 2\log_2 n$), we can conclude that $m_{k+1} \leq 2\log_2 n$ on any such iteration $k+1$. Thus the loop invariant holds.

In all cases, the loop invariant holds.

Hence, by the principle of mathematical induction, we conclude that the number of excluded elements $m \leq 2\log_2 n$. $\square$

5. Use the previous statements to design and prove the correctness and run-time of an algorithm to sort the nearly sorted array in time $O(n)$.

Solution: CloseSort(A) where $A[1, \ldots, n]$ is a close-to-sorted array such that at most $\log_2 n$ elements are out of place:

Let S be an empty stack. Perform Algorithm 1 using A and S, storing each of the ℓ pairs of elements (a, b) excluded from S in ℓ-many 2-element arrays $(a_1, b_1), (a_2, b_2), \ldots, (a_\ell, b_\ell)$ such that $a_i < b_i$ for all $1 \leq i \leq \ell$.

Return $A' \leftarrow$ Merge(S,$(a_1, b_1), (a_2, b_2), \ldots, (a_\ell, b_\ell)$). Here Merge($A_1, A_2, \ldots, A_k$) is the well-known algorithm to merge k-many monotonically increasing sequences into a single sorted array. Note that the running time of Merge is known to be $O(n)$, where n is the sum of the lengths of the k arrays to merge. Intuitively, this is because the algorithm involves scanning each sequences and copying values from the source sequences into the destination array.

Lemma I.

Claim. CloseSort(A) returns an array A' containing the elements of A in sorted order.

Proof. From Invariant I, we have that S is a monotonically increasing subsequence of A.

By construction, each of (a_i, b_i) for $1 \leq i \leq \ell$ is monotonically increasing.

From the Merge invariant, A' contains the elements of S and $(a_1, b_1), (a_2, b_2), \ldots, (a_\ell, b_\ell)$ in sorted order.

By construction, S is the result of Algorithm 1 and the arrays $(a_1, b_1), (a_2, b_2), \ldots, (a_\ell, b_\ell)$ contain all elements in A not included in S, and only those elements in A not included in S. Therefore A' contains all elements in A, and only elements in A. That is, for all $a \in A$, we have $a \in A'$ and for all $a' \in A'$, we have $a' \in A$.

Hence, CloseSort returns an array A' containing the elements of A in sorted order.

Lemma II.

Claim. The running time of CloseSort(A) is $O(n)$.

Proof. Let $T(n)$ denote the running time of CloseSort(A). From Invariant II, we have that the number of excluded elements $m \leq 2 \log_2 n$.

Therefore, there are at most $\log_2 n$ pairs of excluded elements. We have $\ell \leq \log_2 n$.

The running time of CloseSort is the sum of the running times of Algorithm 1, $T_1(n)$, and Merge, $T_2(n)$, plus some constant overhead C. The running times of Algorithm 1 and Merge are given to be linear with respect to their inputs.

$$\begin{aligned} T(n) &= T_1(n) + T_2(n) + C \\ &= O(n) + O(\text{len}(S) + 2\ell) + C && \textit{considering the number of elements to merge,} \\ &= O(n) + O(n) + C && \textit{len}(S) + 2\ell = n \textit{ by construction,} \\ &= O(n) && \textit{asymptotically.} \end{aligned}$$

Hence, the running time of CloseSort on an n-element almost-sorted array is $O(n)$.

Proof.

Let $A[1, \ldots n]$ be a close-to-sorted array such that at most $\log_2 n$ elements are out of place.

From Lemma I, CloseSort(A) returns A', the sorted version of A.

From Lemma II, the running time of CloseSort(A) is $O(n)$.

Ergo, CloseSort sorts a close-to-sorted array in linear time. $\square$