Problem Solutions to CLRS

Zeaiter Zeaiter

Contents

1 Chapter 2

2.1–2

DECREASING-INSERTION-SORT(A)

```
1  for i = 1 to A.length − 1
2      key = A[i]
3      j = i − 1
4      while j > −1 and A[j] < key
5          A[j + 1] = A[j]
6          j = j − 1
7      A[j + 1] = key
```

2.1–3

LINEAR-SEARCH(A, v)

```
1  for i = 0 to A.length − 1
2      if A[i] == v
3          return i
4  return NIL
```

Loop Invariant: At the start of each iteration of the **for** loop (lines 1–4) $i - 1$ is not an index of A such that $A[i-1] = v$.

Proof. Let us now prove the correctness of our algorithm. Suppose $i = 0$, then $i - 1$ is clearly not an index of A and hence $A[i-1]$ is undefined. Now suppose the loop invariant is true for some i, that is, $i - 1$ is not an index of A such that $A[i-1] = v$, or equivalently, $A[i-1] \neq v$. Then at line 3 the **if** loop will **return** i if $A[i] = v$, in which case the **for** loop terminates and there is no further iteration. Otherwise, if $A[i] \neq v$ then at the start of the next for loop iteration $(i + 1) - 1$ is not an index of A such that $A[(i+1)-1] = v$. Finally, for termination to occur we have either $i = n + 1$ where $n = A$.length in which case the algorithm returns NIL indicating v is not an element of A. Otherwise, termination occurs because of the nested **if** on line 3 which causes the algorithm to return i which indicates the index of A such that $A[i] = v$. □

2.1–4

Input: Two sequences of n integers, $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, such that $0 \leq a_i, b_i \leq 1$ for $i = 1, \ldots, n$. Least significant digits are first.

Output: An array $C = (c_1, \ldots, c_n, c_{n+1})$ such that $0 \leq c_i \leq 1$ for $i = 1, \ldots, n+1$ and $C' = A' + B'$ where $\cdot'$ is the integer represented by $\cdot$.

BINARY-ADDITION(A, B)

```
1  define integer C[A.length + 1]
2  overflow = 0
3  for i = 0 to A.length − 1
4      C[i] = (A[i] + B[i] + overflow) % 2
5      overflow = (A[i] + B[i] + overflow)/2
6  C[i] = overflow
7  return C
```

2.2–1

The function is $O(n^3)$

2.2–2

SELECTION-SORT(A)		cost	times
1	**for** $i = 0$ **to** A.length $- 2$	c_1	n
2	min $= i$	c_2	$n - 1$
3	**for** $j = i + 1$ **to** A.length-1	c_3	$\sum_{i=0}^{n}(n - i + 1)$
4	**if** $A[j] < A[\text{min}]$	c_4	$\sum_{i=0}^{n}(n - i)$
5	min $= j$	c_5	$\sum_{i=0}^{n} t_i$
6	$M = A[\text{min}]$	c_6	$n - 1$
7	$A[\text{min}] = A[i]$	c_7	$n - 1$
8	$A[i] = M$	c_8	$n - 1$

Loop Invariant: At the start of each iteration of the **for** loop (lines 1–8) the sub-array $A[0 \ldots i]$ is sorted in non-decreasing order.

The algorithm only needs to run for the first $n-1$ elements since this will arrange the $n - 1$ smallest elements in non-decreasing order, ensuring the n^{th} element at the end is in the appropriate position. That is, $A[n] \geq A[i]$ for $i = 0, \ldots, n - 2$.

The best-case running time occurs when the given array is already sorted from smallest to largest. In such a case $t_i = 0$ since we never need to re-assign the

minimum index. The runtime equation is,

$$\begin{aligned}
T(n) &= c_1 n + (c_2 + c_6 + c_7 + c_8)(n-1) + c_3 \sum_{i=0}^{n}(n-i+1) + c_4 \sum_{i=0}^{n}(n-i) \\
&= c_1 n + (c_2 + c_6 + c_7 + c_8)(n-1) + c_3 \left((n+1) + \frac{n}{2}(n+1)\right) + c_4 \left(n + \frac{n}{2}(n-1)\right) \\
&= (c_3 + c_4)\frac{n^2}{2} + (c_1 + c_2 + c_6 + c_7 + c_8 + \frac{3}{2}c_3 + \frac{1}{2}c_4)n + (c_2 + c_6 + c_7 + c_8 + c_3)
\end{aligned}$$

and so the best-case running time is $O(n^2)$. In a worst-case scenario, the array given to the procedure is in descending order, however this would only include an additional term to $T(n)$ above,

$$c_5 \sum_{i=0}^{n}(n-1) = c_5 \left(n + \frac{n}{2}(n-1)\right) = \frac{1}{2}c_5(n^2 + n)$$

since here line 5 will re-assign the minimum for all remaining entries in the array. So the runtime in a worst-case scenario is also $O(n^2)$.

2.2–3

LINEAR-SEARCH(A, v)	cost	times
1 **for** $i = 0$ **to** A.length $- 1$	c_1	$n + 1$
2 **if** $A[i] == v$	c_2	n
3 **return** i	c_3	t_1
4 **return** NIL	c_4	t_2

If each of the n elements of A have equal probability p to be v then the expected value is,

$$E[v] = 0 \times \frac{1}{n} + 1 \times \frac{1}{n} + 2 \times \frac{1}{n} + \cdots + n \times \frac{1}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n}{2}(n+1) = \frac{n+1}{2}$$

and hence on average we need to search through $\frac{n+1}{2}$ elements to find v. In the worst case we need to search n elements since v is not present in A. We have the following runtime equation,

$$T(n) = c_1(n+1) + c_2 n + c_3 t_1 + c_4 t_2$$

In the average-case $t_1 = \frac{1}{2} = t_2$ then,

$$T(n) = (c_1 + c_2)n + c_1 + \frac{1}{2}(c_3 + c_4)$$

and so the runtime is $O(n)$. In the worst-case $t_1 = 0$ and $t_2 = 1$ so the runtime equation is,

$$T(n) = (c_1 + c_2)n + c_1 + c_3$$

and so we still have $O(n)$ runtime.

2.2–4

Implement a checking loop/statement to return the procedure if in a best-case scenario. For example in Selection-Sort we can implement an initial loop that checks if the given array is already in sorted order and then return,

		cost	times
1	**for** $i = 0$ **to** A.length $- 2$	c_1	n
2	**if** $A[i] > A[i+1]$	c_2	$n-1$
3	**break**	c_3	t_1
4	**if** $i ==$ A.length $- 2$	c_4	1
5	**return**	c_5	t_2

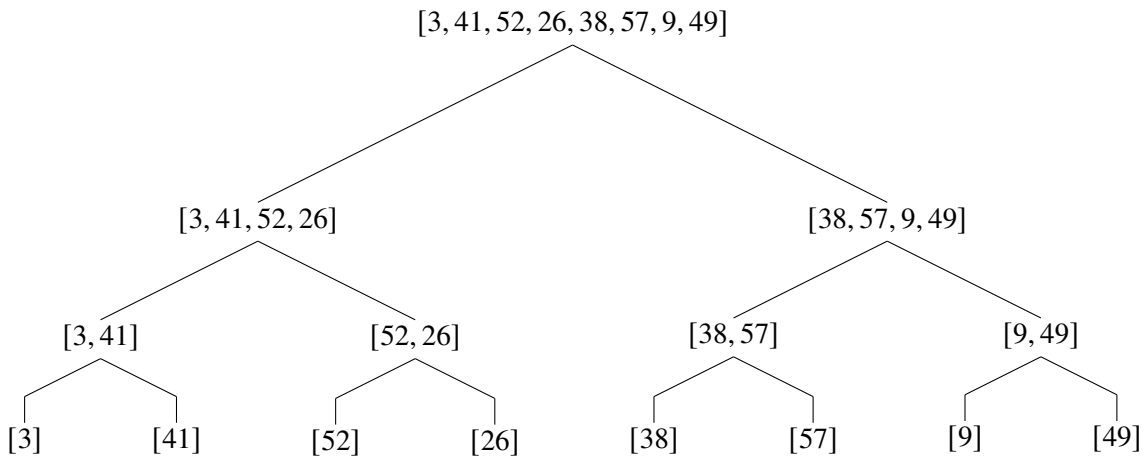
In such a case the runtime will be,

$$T(n) = (c_1 + c_2)n - c_2 + c_4 + c_5$$

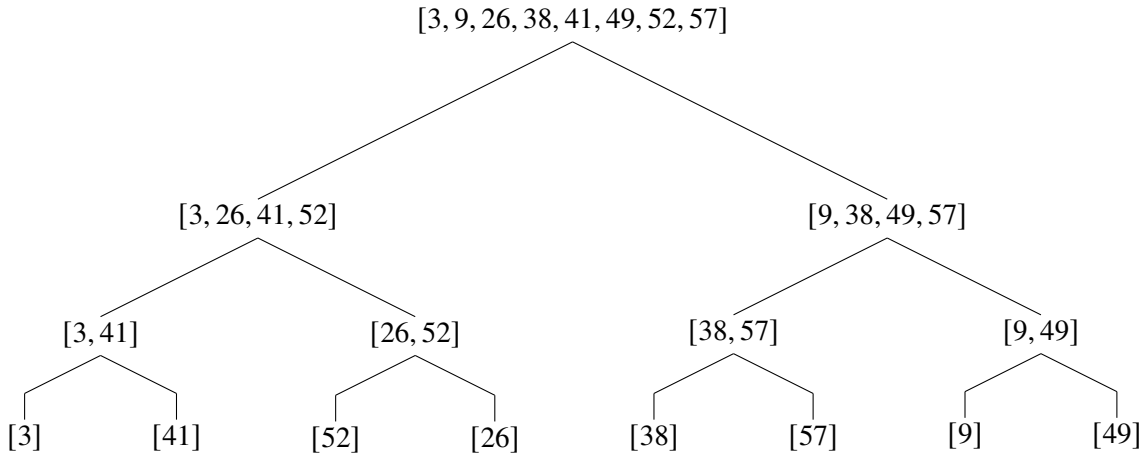
which is $O(n)$ a significant improvement over $O(n^2)$ in the above exercise.

2.3–1

We first divide the array into sub-arrays until we have arrays of length 1.



Then we merge to eventually recover the original array in sorted order.



2.3–2

MERGE(A, p, q, r)

```
 1  n_1 = q − p
 2  n_2 = r − q − 1
 3  define integers L[0 . . . n_1] and R[0 . . . n_2]
 4  for i = 0 to n_1
 5      L[i] = A[p + i]
 6  for j = 0 to n_2
 7      R[j] = A[q + j + 1]
 8  i = 0
 9  j = 0
10  for k = p to r
11      if i > n_1
12          A[k] = R[j]
13          j = j + 1
14      elseif j > n_2
15          A[k] = L[i]
16          i = i + 1
17      elseif L[i] ≤ R[j]
18          A[k] = L[i]
19          i = i + 1
20      else
21          A[k] = R[j]
22          j = j + 1
```

2.3–3

Proposition 1.1. *If $n = 2^k$ for $k \in \mathbb{N} \setminus \{0\}$ then the solution of,*

$$T(n) = \begin{cases} 2 & \text{if } k = 1 \\ 2T(n/2) + n & \text{if } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

Proof. If $k = 1$ we have $n = 2$ so $T(2) = 2 = 2 \lg 2$. Now assume this is true for some $k = m > 1$ then $T(2^m) = 2^m \lg 2^m$ so for 2^{m+1} we have the recurrence,

$$\begin{aligned} T(2^{m+1}) = 2T(2^{m+1}/2) + 2^{m+1} &= 2T(2^m) + 2 \cdot 2^m \\ &= 2 \cdot 2^m \lg 2^m + 2 \cdot 2^m \\ &= 2^{m+1} (\lg 2^m + 1) \\ &= 2^{m+1} (\lg 2^m + \lg 2) = 2^{m+1} \lg 2^{m+1} \end{aligned}$$

Hence the solution for any $n = 2^k$, $k \in \mathbb{N} \setminus \{0\}$, is $T(n) = n \lg n$. □

2.3–4

Let $T(n)$ be the time to sort an array of length n and $I(n)$ be the time to insert an element into an array of length n.

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(n-1) + I(n-1) & \text{otherwise} \end{cases}$$

2.3–5

BINARY-SEARCH(A, v)

```
1  mid = ⌊A.length/2⌋
2  if A[mid] == v
3      return mid
4  elseif mid == 0
5      return FALSE
6  elseif A[mid] < v
7      BINARY-SEARCH(A[mid ... A.length − 1], v)
8  else
9      BINARY-SEARCH(A[0 ... mid], v)
```

The running time of this algorithm consists of the re-running binary-search on arrays of approximate length $n/2$ twice and checking if the middle value of the total array is the target value, v. We can express this as the recurrence,

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(n/2) + c & \text{otherwise} \end{cases}$$

The worst case scenario occurs when A does not contain v in which case we can expand the recurrence as,

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c \\ &= T\left(\frac{n}{2 \cdot 2}\right) + (1+1)c \\ &= T\left(\frac{n}{2 \cdot 2 \cdot 2}\right) + (1+1+1)c \\ &\vdots \\ &= T\left(\frac{n}{2^{\lg n+1}}\right) + (\lg n + 1)c \approx T(1) + (\lg n + 1)c \end{aligned}$$

So after $\lg n + 1$ iterations the algorithm returns NIL and from above we have that the runtime is $O(\lg n)$.

2.3–6

No, this is not possible since insertion-sort will still need to shift $i - 1$ elements which will always create $O(n^2)$ time in the worst case.

2.3–7

We first need to sort S in ascending order. We can then determine whether there are two elements of S that sum to x by performing a binary-search for $x - S[i]$ in S.

```
SUM-DECOMPOSITION(S, x)
1  S = MERGE-SORT(S, 0, S.length)
2  for i = 0 to S.length
3      if BINARY-SEARCH(S, x − S[i]) ≠ NIL
4          return TRUE
5  return FALSE
```

From earlier in the chapter merge-sort will at worst take $O(n \lg n)$ and from 2.3–5 binary-search is at worst $O(\lg n)$. However as we loop on binary-search the worst-case for the loop on lines 2–4 will be $O(n \lg n)$. Hence, sum-decomposition has a worst-case runtime of $O(n \lg n)$.

Resource Models

When we consider a model for analysing the time complexity of an algorithm, such as the ***random-access machine (RAM)*** model we need to define a word size of data. Here *word* indicates some object to be stored in data. This topic is important because it creates a limit on how much information can be stored in a single word. If we do not make such assumptions then arguably one can store an infinite amount of data in each word and so every algorithm has constant runtime. Clearly this cannot be true. For our purposes when we want to work with inputs of size n we must have that each word of data can store the value n and so the integers are represented by $c \lg n$ bits for some $c \geq 1$.

This representation in bits can be seen as follows. Since lg is base 2 we then have $2^{\lg n} = n$. However note that in machine counting we start at 0 so if we have $\lg n$ bits in a machine we can count from 0 to $n - 1$. Hence, we require at least one extra bit, that is, $\lg n + 1$ bits. This can also be expresses as $c \lg n$ bits for some $c \geq 1$.

The importance of this calculation becomes more obvious when we deal with recurrences. For example, if we start with an input of size n and our recurrence halves the input size at each iteration,

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n/2) & \text{otherwise} \end{cases}$$

Then we know that we will need at least $\lg n$ iterations to reach a constant case as,

$$T\left(\frac{n}{2 \cdot 2 \cdots 2}\right) = T\left(\frac{n}{2^{\lg n}}\right) \approx T(1)$$

Remember, we are simulating counting in a machine which starts from 0 and hence the $\approx$ above instead of $=$.