# Problem Solutions to CLRS

Zeaiter Zeaiter

## Contents

# 1 Getting Started

## 1.1 Insertion Sort - Exercises

### 2.1–2

```
DECREASING-INSERTION-SORT(A)
for i = 1 to A.length − 1
    key = A[i]
    j = i − 1
    while j > −1 and A[j] < key
        A[j + 1] = A[j]
        j = j − 1
    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$(A, B)$

**define** integer $C[A.\text{length} + 1]$<br>
overflow = 0<br>
**for** $i = 0$ **to** $A$.length − 1<br>
    $C[i] = (A[i] + B[i] + \text{overflow}) \,\%\, 2$<br>
    overflow = $(A[i] + B[i] + \text{overflow})/2$<br>
$C[i]$ = overflow<br>
**return** $C$

## 1.2 Analysing Algorithms - Exercises

### 2.2–1

The function is $\Theta(n^3)$

### 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 $n^{\text{th}}$ 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 $\Theta(n^2)$. 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 $\Theta(n^2)$.

**2.2–3**

| LINEAR-SEARCH$(A, v)$ | cost | times |
|---|---|---|
| **for** $i = 0$ **to** $A$.length $- 1$ | $c_1$ | $n+1$ |
|     **if** $A[i] == v$ | $c_2$ | $n$ |
|         **return** $i$ | $c_3$ | $t_1$ |
| **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 $\Theta(n)$. 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 $\Theta(n)$ 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 |
|---|---|---|
| **for** $i = 0$ **to** $A$.length $- 2$ | $c_1$ | $n$ |
|     **if** $A[i] > A[i+1]$ | $c_2$ | $n-1$ |
|         **break** | $c_3$ | $t_1$ |
| **if** $i == A$.length $- 2$ | $c_4$ | 1 |
|     **return** | $c_5$ | $t_2$ |

In such a case the runtime will be,

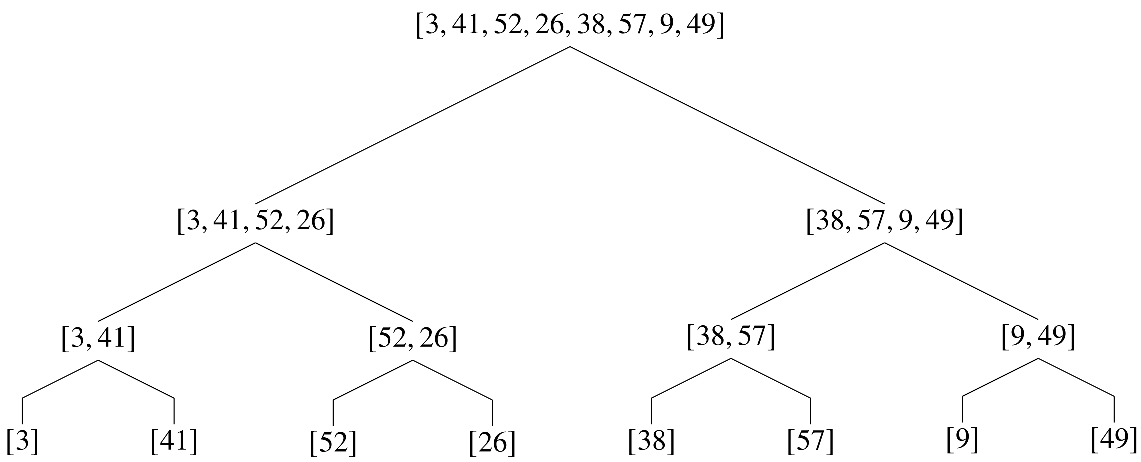
$$T(n) = (c_1 + c_2)n - c_2 + c_4 + c_5$$

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

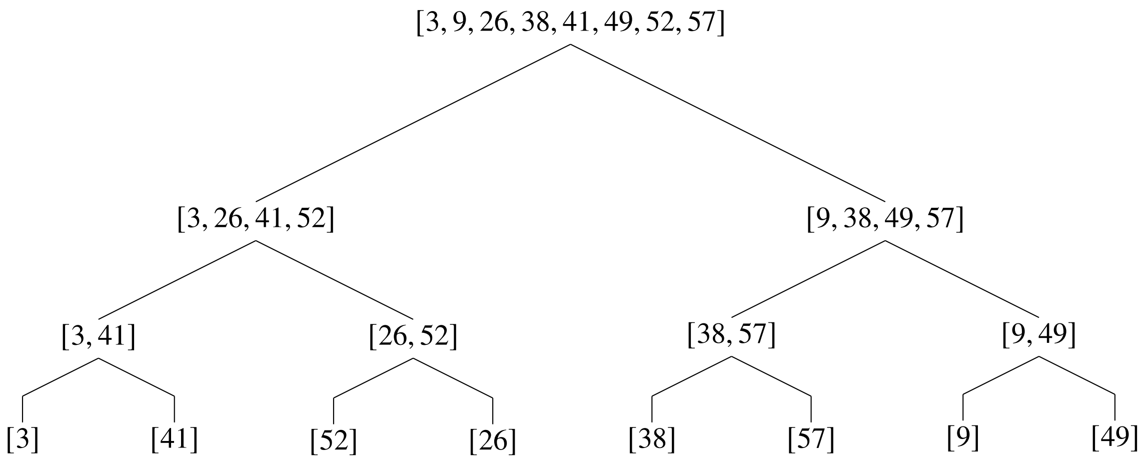
## 1.3 Designing Algorithms - Exercises

### 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)
n1 = q − p
n2 = r − q − 1
define integers L[0 . . . n1] and R[0 . . . n2]
for i = 0 to n1
      L[i] = A[p + i]
for j = 0 to n2
      R[j] = A[q + j + 1]
i = 0
j = 0
for k = p to r
      if i > n1
            A[k] = R[j]
            j = j + 1
      elseif j > n2
            A[k] = L[i]
            i = i + 1
      elseif L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
      else
            A[k] = R[j]
            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 $T(n) = n \lg n$. $\square$

### 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 $n$.

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

### 2.3–5

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

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

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 $\Theta(\lg n)$.

**2.3–6**

No, this is not possible since insertion-sort will still need to shift $i - 1$ elements which will always create $\Theta(n^2)$ 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 $\Theta(n \lg n)$ and from 2.3–5 binary-search is at worst $\Theta(\lg n)$. However as we loop on binary-search the worst-case for the loop on lines 2–4 will be $\Theta(n \lg n)$. Hence, sum-decomposition has a worst-case runtime of $\Theta(n \lg n)$.

# 2 Growth of Function

We will make clear some asymptotic notation used in the text.

**Definition 2.1.** Given functions $f, g : \mathbb{N} \to \mathbb{R}$ we say $f(n) = \Theta(g(n))$ if there exists $c_1, c_2 > 0$ and $n_0 \in \mathbb{N}$ such that $c_2 |g(n)| \leq |f(n)| \leq c_1 |g(n)|$. More generally, we consider $\Theta(g(n))$ to be the set of all functions that satisfy the above statement.

**Definition 2.2.** Given functions $f, g : \mathbb{N} \to \mathbb{R}$ we say $f(n) = O(g(n))$ if there exists $c > 0$ and $n_0 \in \mathbb{N}$ such that $|f(n)| \leq c |g(n)|$ for all $n \geq n_0$. More generally, we consider $O(g(n))$ to be the set of all functions that satisfy the above statement.

**Definition 2.3.** Given functions $f, g : \mathbb{N} \to \mathbb{R}$ we say $f(n) = \Omega(g(n))$ if there exists $c > 0$ and $n_0 \in \mathbb{N}$ such that $c |g(n)| \leq |f(n)|$ for all $n \geq n_0$. More generally, we consider $\Omega(g(n))$ to be the set of all functions that satisfy the above statement.

Notice that in all three of the above definitions we can consider the defined structure as a set or a identity between two functions. In this case we can write either $f \in O(g(n))$ or $f = O(g(n))$. Arguably the correct notation is $f \in O(g(n))$ as this works for the most general case where we consider $O(g(n))$ to be a set. However, we will often use the equality notation as a matter of simplicity since the function we compare with remains constant and we wish to make a statement about a particular element of the set in question.

We have the following two definitions which are more restrictive than those above.

**Definition 2.4.** Given functions $f, g : \mathbb{N} \to \mathbb{R}$ we say $f(n) = o(g(n))$ if for every $c > 0$ there exists $n_0 \in \mathbb{N}$ such that $|f(n)| < c |g(n)|$ for all $n \geq n_0$. More generally, we consider the set $o(g(n))$ to be the set of all functions that satisfy the above statement.

**Definition 2.5.** Given functions $f, g : \mathbb{N} \to \mathbb{R}$ we say $f(n) = \omega(g(n))$ if for any positive constant $c > 0$ there exists $n_0 \in \mathbb{N}$ such that $c |g(n)| < |f(n)|$ for all $n \geq n_0$. More generally, we consider $\omega(g(n))$ to be the set of all functions that satisfy the above statement.

## 2.1 Asymptotic Notation - Exercises

### 3.1–1

Suppose $f, g : \mathbb{N} \to \mathbb{R}$ are asymptotically non-negative. Then there exists $n_f, n_g \in \mathbb{N}$ such that,

$$0 \leq f(n) \quad \forall n \geq n_f$$
$$0 \leq g(n) \quad \forall n \geq n_g$$

If we take $n_0 := \max(n_f, n_g)$ then,

$$0 \leq f(n), g(n) \quad \forall n \geq n_0$$

and so,

$$0 \leq \max(f(n), g(n)) \quad \forall n \geq n_0$$

By definition,

$$\max(f(n), g(n)) = \begin{cases} f(n) & \text{if } f(n) \geq g(n) \\ g(n) & \text{otherwise} \end{cases}$$

so by the asymptotic non-negativity, $\max(f(n), g(n)) \leq f(n)+g(n)$ and $\max(f(n), g(n)) \geq \frac{f(n)+g(n)}{2}$ for all $n \geq n_0$. Hence, $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$.

**3.1–2**

Let $a, b \in \mathbb{R}$ such that $b > 0$. Choose $n_0 \in \mathbb{N}$ such that $n_0 \geq \lceil |a| \rceil$ then,

$$(n + a)^b \leq 2^b n^b \quad \forall n \geq n_0$$

so $(n + a)^b = O(n^b)$. Working in reverse,

$$\begin{aligned} (n + a)^b &\geq cn^b \\ n + a &\geq c^{\frac{1}{b}} n \\ n - c^{\frac{1}{b}} n &\geq -a \\ n &\geq \frac{-a}{1 - c^{\frac{1}{b}}} \end{aligned}$$

it then suffices to choose $c < 1$. So for any $c = \frac{1}{2}$ and $n_0 \geq \frac{-a}{1-c^{\frac{1}{b}}}$ we then have,

$$(n + a)^b \geq cn^b$$

hence, $(n + a)^b \in \Omega(n^b)$. Therefore, $(n + a)^b \in \Theta(n^b)$.

**3.1–3**

The statement "The running time of the algorithm $A$ is at least $O(n^2)$" is meaningless as $O(n^2)$ gives an asymptotic upper bound, or in terms of algorithms, the worst-case running time. Since $f(n) = O(n^2)$ means there exists $c > 0$ and $n_0 \in \mathbb{N}$ such that,

$$0 \leq f(n) \leq cn^2 \quad \forall n \geq n_0$$

It is clear from the definition that the algorithm runtime may or may not be less than $cn^2$.

### 3.1–4

(a) Given $c \geq 2$ we can write $2^{n+1} = 2 \cdot 2^n \leq c2^n$ for any $n \in \mathbb{N}$. So $2^{n+1} = O(2^n)$.

(b) Suppose there exists $c > 0$ and $n_0 \in \mathbb{N}$ such that $0 \leq 2^{2n} \leq c2^n$ for all $n \geq n_0$.Then $0 \leq 2^n \leq c$ for all $n \geq n_0$ which is clearly a contradiction. Hence, $2^{2n} \neq O(2^n)$.

### 3.1–5

**Theorem 2.6.** *For any two functions $f, g : \mathbb{N} \to \mathbb{R}$ we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

*Proof.* First suppose $f(n) = \Theta(g(n))$, then by definition there exists $c_1, c_2 > 0$ and $n_0 \in \mathbb{N}$ such that,

$$c_2 |g(n)| \leq |f(n)| \leq c_1 |g(n)| \quad \forall n \geq n_0$$

and so $f(n)$ satisfies the conditions of $O(g(n))$ and $\Omega(g(n))$.

Now consider $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Then, there exists $c_1, c_2 > 0$ and $n_1, n_2 \in \mathbb{N}$ such that,

$$\begin{aligned} |f(n)| \leq c_1 |g(n)| &\quad \forall n \geq n_1 \\ c_2 |g(n)| \leq |f(n)| &\quad \forall n \geq n_2 \end{aligned}$$

Set $n_0 := \max(n_1, n_2)$ then we have,

$$c_2 |g(n)| \leq |f(n)| \leq c_1 |g(n)| \quad \forall n \geq n_0$$

and so $f(n) = \Theta(g(n))$. □

### 3.1–6

**Corollary 2.7.** *The running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.*

*Proof.* This is an immediate application of Theorem 2.6. Let $W(n)$ denote the worst-case running time, $B(n)$ denote the best-case running time and $T(n)$ denote the running time of the algorithm. Then,

$$B(n) \leq T(n) \leq W(n) \quad \forall n \in \mathbb{N}$$

and so $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$. Hence, by Theorem 2.6 $T(n) = \Theta(g(n))$. □

**3.1–7**

**Proposition 2.8.** *The set $o(g(n)) \cap \omega(g(n))$ is empty.*

*Proof.* It is sufficient to show that for any $f(n) = o(g(n))$ we have $f(n) \neq \omega(g(n))$. Suppose $f(n) = o(g(n))$ then for every $c > 0$ there exists $n_0 \in \mathbb{N}$ such that $|f(n)| < c\,|g(n)|$ for all $n \geq n_0$. Assume $f(n) = \omega(g(n))$ then for any $c > 0$ we can choose $n_0 \in \mathbb{N}$ so that,

$$c\,|g(n)| < |f(n)| < c\,|g(n)|$$

This is clearly a contradiction and so we have $f(n) \neq \omega(g(n))$. Hence, $o(g(n)) \cap \omega(g(n)) = \emptyset$. □

**3.1–8**

$$\begin{aligned}
O(g(n,m)) := \{f(n,m) \colon\ & \exists c > 0 \text{ and } \exists n_0, m_0 \in \mathbb{N} \text{ such that} \\
& |f(n,m)| \leq c\,|g(n,m)|\ \forall n \geq n_0 \text{ or } m \geq m_0\} \\
\Omega(g(n,m)) := \{f(n,m) \colon\ & \exists c > 0 \text{ and } \exists n_0, m_0 \in \mathbb{N} \text{ such that} \\
& c\,|g(n,m)| \leq |f(n,m)|\ \forall n \geq n_0 \text{ or } m \geq m_0\} \\
\Theta(g(n,m)) := \{f(n,m) \colon\ & \exists c_1, c_2 > 0 \text{ and } \exists n_0, m_0 \in \mathbb{N} \text{ such that} \\
& c_2\,|g(n,m)| \leq |f(n,m)| \leq c_1\,|g(n,m)|\ \forall n \geq n_0 \text{ or } m \geq m_0\}
\end{aligned}$$

## 2.2 Standard Notations and Common Functions - Exercises

**3.2–1**

**Proposition 2.9.** *Let $f, g : \mathbb{N} \to \mathbb{N}$ such that $f$ and $g$ are monotonically increasing. Then $f(n) + g(n)$, $f(g(n))$ and $g(f(n))$ are monotonically increasing. Moreover, if $f$ and $g$ are non-negative then $f(n) \cdot g(n)$ is monotonically increasing.*

*Proof.* Clearly,

$$f(n) + g(n) \leq f(m) + g(m)$$

for all $n, m \in \mathbb{N}$ such that $n \leq m$. Now if $n, m \in \mathbb{N}$ so that $n \leq m$ then,

$$g(n) \leq g(m) \implies f(g(n)) \leq f(g(m))$$

Likewise the result will be true for $g(f(n))$. If $f$ and $g$ are non-negative then $f(n) \cdot g(n)$ is non-negative for all $n \in \mathbb{N}$. So for $n, m \in \mathbb{N}$ such that $n \leq m$,

$$f(n) \cdot g(n) \leq f(m) \cdot g(n) \leq f(m) \cdot g(m)$$

□

*Remark* 2.10. The results of Proposition 2.9 are strict if $f$ and $g$ are strictly monotonically increasing.

### 3.2–2

We write,

$$a^{\log_b c} = a^{\frac{\log_a c}{\log_a b}} = \left(a^{\log_a c}\right)^{\frac{1}{\log_a b}} = c^{\frac{1}{\log_a b}}$$

Now,

$$\frac{1}{\log_a b} = \frac{\log_b a}{\log_b b} = \log_b a$$

Hence, from above,

$$a^{\log_b c} = c^{\frac{1}{\log_a b}} = c^{\log_b a}$$

### 3.2–3

Stating Sterling's approximation,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

we then have,

$$\begin{aligned}
\lg n! &= \lg\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)\right) \\
&= \lg\sqrt{2\pi n} + n\lg\frac{n}{e} + \lg\left(1 + \Theta\left(\frac{1}{n}\right)\right) \\
&= \frac{1}{2} + \frac{1}{2}\lg\pi + \frac{1}{2}\lg n + n\lg\frac{1}{e} + n\lg n + \lg\left(1 + \Theta\left(\frac{1}{n}\right)\right) \\
&= \Theta(1) + \Theta(1) + \Theta(\lg n) + \Theta(n) + \Theta(n\lg n) + \lg\Theta(1) = \Theta(n\lg n)
\end{aligned}$$

Now,

$$n! = n \cdot (n-1)\cdots 2 \cdot 1 < n \cdot n \cdots n \cdot n = n^n \quad \forall n \in \mathbb{N}$$

So for $c \geq 1$ we have the desired property. For $0 < c < 1$ choose $n_0$ so that $cn_0 > 1$ then,

$$n! = n \cdot (n-1)\cdots 2 \cdot 1 < n \cdot (n-1)\cdots 2 \cdot cn < cn^n \quad \forall n \geq n_0$$

Hence, $n! = o(n^n)$.

Finally,

$$n! = n \cdot (n-1)\cdots 2 \cdot 1 = n \cdot (n-1)\cdots 2 > 2^{n-1} = \frac{1}{2}2^n \quad \forall n \in \mathbb{N}$$

So for $c \leq \frac{1}{2}$ we have the desired property. For $c > \frac{1}{2}$ choose $n_0$ so that $n_0 > c2^2$ so,

$$n! = n \cdot (n-1)\cdots 2 > c2^2 \cdot (n-1)\cdots 2 > c2^n \quad \forall n \geq n_0$$

Hence $n! = \Omega(2^n)$.

### 3.2–4

Assume $\lceil \lg n \rceil ! = O(n^m)$ for some $m \in \mathbb{N}$. Then there exists $c > 0$ and $n_0 \in \mathbb{N}$ such that $0 \le \lceil \lg n \rceil ! \le cn^m$ for all $n \ge n_0$. From exercise 3.2–3 we have $n! = \omega(2^n)$ so for any fixed $n \ge n_0$ and for any $\varepsilon > c(2^n)^m$ there exists $N_0 \ge n_0$ such that,

$$c(2^n)^m 2^n < \varepsilon 2^n < n! = \lceil \lg 2^n \rceil ! \le c(2^n)^m \quad \forall n \ge N_0$$

A contradiction and so $\lceil \lg n \rceil ! \ne O(n^m)$.

For any $n \in \mathbb{N}$ such that $n \ge 2$ take $k = \lg \lg n$ so that we can write $n = 2^{2^k}$ then $\lceil \lg \lg n \rceil ! \le n^2$, for some $m \in \mathbb{N}$, is equivalent to $\lceil k \rceil ! \le 2^{2^{k^2}}$. Setting $m = \lceil k \rceil$ we have

$$\begin{aligned} \lceil k \rceil ! = m! = m \cdot (m-1) \cdots 2 \cdot 1 &\le 2^m \cdot 2^m \cdots 2^m \cdot 2^m \\ &\le 2^{m^2} \le 2^{2^{m^2}} \le 2^{2^{k^2}} \end{aligned}$$

Hence, $\lceil \lg \lg n \rceil ! = O(n^2)$.

### 3.2–5

By definition $\lg^* n := \min \left\{ i \ge 0 \colon \ \lg^{(i)} n \le 1 \right\}$ where,

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i)}(n)) & \text{if } i > 0 \end{cases}$$

For any $n \ge 1$ take $k = \lg n$ then $n = 2^k$ and so $\lg^* 2^k = 1 + \lg^* k$ and $\lg^*(\lg 2^k) = \lg^* k$. Then,

$$\frac{\lg(\lg^* 2^k)}{\lg^*(\lg 2^k)} = \frac{\lg(1 + \lg^* k)}{\lg^* k}$$

By an application of L'hopital's rule we can consider the quotient above to be a subsequence of the real valued quotient, in which case,

$$\lim_{k \to \infty} \frac{\lg(1 + \lg^* k)}{\lg^* k} = \lim_{k \to \infty} \frac{1}{1 + \lg^* k} \to 0$$

So $\lg^*(\lg n)$ is asymptotically larger.

### 3.2–6

We have,

$$\varphi = \frac{1 + \sqrt{5}}{2} \qquad \hat{\varphi} = \frac{1 - \sqrt{5}}{2}$$

Then,

$$\varphi^2 = \frac{6 + 2\sqrt{5}}{4} = \frac{3 + \sqrt{5}}{2} = \frac{1 + \sqrt{5} + 2}{2} = \varphi + 1$$

and,

$$\hat{\varphi}^2 = \frac{6 - 2\sqrt{5}}{2} = \frac{3 - \sqrt{5}}{2} = \frac{1 - \sqrt{5} + 2}{2} = \hat{\varphi} + 1$$

**3.2–7**

Suppose $n = 0$ then,

$$\frac{\varphi^0 - \hat{\varphi}^0}{\sqrt{5}} = \frac{0}{\sqrt{5}} = F_0$$

If $n = 1$,

$$\frac{\varphi - \hat{\varphi}}{\sqrt{5}} = \frac{\frac{2\sqrt{5}}{2}}{\sqrt{5}} = 1 = F_1$$

Assume that for some $n$ we have,

$$F_n = \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}$$

then,

$$\begin{aligned} F_{n+1} = F_n + F_{n-1} &= \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}} + \frac{\varphi^{n-1} - \hat{\varphi}^{n-1}}{\sqrt{5}} \\ &= \frac{\varphi^n + \varphi^{n-1} - \hat{\varphi}^n - \hat{\varphi}^{n-1}}{\sqrt{5}} \\ &= \frac{\varphi^{n-1}(\varphi + 1) - \hat{\varphi}^{n-1}(\hat{\varphi} + 1)}{\sqrt{5}} \\ &= \frac{\varphi^{n-1}\varphi^2 - \hat{\varphi}^{n-1}\hat{\varphi}^2}{\sqrt{5}} \\ &= \frac{\varphi^{n+1} - \hat{\varphi}^{n+1}}{\sqrt{5}} \end{aligned}$$

**3.2–8**

If $k \lg k = \Theta(n)$ then there exists $c_1, c_2 > 0$ and $n_0, k_0 \in \mathbb{N}$ such that,

$$0 \leq c_2 n \leq k \lg k \leq c_1 n \quad \forall n \geq n_0, \forall k \geq k_0$$

which gives,

$$\lg c_2 + \lg n \leq \lg k + \lg\lg k \leq \lg c_1 + \lg n$$

for all $n \geq n_0$ and $k \geq k_0$. So for $n \geq n_0$ sufficiently large such that $\lg c_1 \leq \lg n$ we have,

$$\lg k \leq \lg k + \lg \lg k \leq \lg c_1 + \lg n \leq 2 \lg n$$

and hence $\lg k = O(\lg n)$. Next,

$$\lg n \leq \lg c_2 + \lg n \leq \lg k + \lg \lg k \leq 2 \lg k$$

and so $\lg k = \Omega(\lg n)$. By Theorem 2.6 we then have that $\lg k = \Theta(\lg n)$. Let $a_1, a_2 > 0$ and $n_0, k_0 \geq 0$ be such that,

$$0 \leq a_2 \lg n \leq \lg k \leq a_1 \lg n \implies \frac{1}{a_2 \lg n} \geq \frac{1}{\lg k} \geq \frac{1}{a_1 \lg n}$$

so from the first inequality above if we take $n_0$ such that for all $n \geq n_0$ we have $\lg c_1 \leq \lg n$ then,

$$c_2 \frac{n}{\lg k} \leq k \leq c_1 \frac{n}{\lg k} \implies \frac{c_2 n}{a_1 \lg n} \leq k \leq \frac{c_1 n}{a_2 \lg n}$$

for all $n \geq n_0$ and $k \geq k_0$. Therefore, $k = \Theta\left(\frac{n}{\lg n}\right)$.

# 3 Elementary Data Structures

The concept of data structures in computer science closely follows that of sets in mathematics. In principle, a data structure is an abstraction of what is referred to as a *dynamic set*.

**Definition 3.1.** A dynamic set is an extension of the mathematical notion of a set where we allow the set to be manipulated.

Typical implementations of dynamic sets involve the set elements representing objects, that is pointers. As we do with pointers, a pointer belonging to a set is synonymous with the object it references belonging to the set. The manipulation of a dynamic set are prescribed operations generally grouped into *queries* and *modifiers*. A list of such typical operations is the following. Let $S$ denote a dynamic set, $x$ a member of $S$ and $k$ a key value:

- $\textsc{Search}(S, k)$
  Return a pointer $x$ in $S$ such that $x.\text{key} = k$, or *nil* if no such element belongs to $S$.

- $\textsc{Insert}(S, x)$
  Add the pointer $x$ to the set $S$. We assume taht any attributes in the object that $x$ references have been initialised in $S$.

- $\textsc{Delete}(S, x)$
  Remove the pointer (and its referenced object) from the set $S$.

- $\textsc{Minimum}(S)$
  Return the pointer $x$ such that $x$ contains the largest key value of all members of $S$.

- $\textsc{Maximum}(S)$
  Return the pointer $x$ such that $x$ contains the smallest key value of all members of $S$.

- $\textsc{Successor}(S, x)$
  Return a pointer $y$ in $S$ such that $y$ is the next largest element in $S$, relative to $x$. If $x$ is the maximum then NIL is returned.

- $\textsc{Predecessor}(S, x)$
  Return a pointer $y$ in $S$ such that $y$ is the next smallest element in $S$, relative to $x$. If $x$ is the minimum then NIL is returned.

## 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 need to be able to index up to $n$. This leads us to require that a word of data must be able to store the numerical value $n$. This is essentially determining how many bits we require in our machine to be able to store a word with numerical values up to, and including, $n$.

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 $n-1$ (equivalently 1 to $n$). Remember that $n$ is arbitrary, however it cannot be varied after we set it in our model. So while our machine may recieve data made up of more than one words, for instance $n^c$ in size, we can still index this data in constant time since $n^c = 2^{c \lg n}$. Meaning that $c \lg n$ operations can be performed in constant time.

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 $=$.