Contents

Chapter 1

Big O

Problem VI.1. In pseudocode and Python:

```
function PRODUCT(a, b)
    sum = 0
    for $i = 0$ to $b - 1$ do
        sum = sum + a
    end for
    return sum
end function
```

```
def product(a, b):
        summ = 0
        for i in range(b):
                summ += a
        return summ
```

The work done within the look is constant time, and the loop is run b times, and therefore the runtime is O(b).

Problem VI.2. In pseudocode and Python:

```
function POWER(a, b)
    if b< 0 then
        return 0
    else if b = 0 then
        return 1
    else
        return a*power(a)
        return a*power(a, b-1)
    end if
end function
```

```
def power(a, b)
        if b < 0:
                return 0
        elif b == 0:
                return 1
        else:
                return a*power(a, b - 1)
```

This is a recursive algorithm to compute a^b. The run time time can be described by the recursion relation $T(b) = T(b-1)+c$, where c is the constant amount of work representing the if statements as well as the multiplication of a by power(a, b-1). As specified in the algorithm, the base case is $T(0) = 1$, which we can use to solve for the fun time:

$$\begin{aligned} T_b &= T(b-1)+c \\ &= T(b-2)+c+c \\ &\vdots \\ &= T(b-b)+bc \\ &= O(1)+O(b) \\ &= O(b). \end{aligned}$$

Problem VI.3. In pseudocode and Python:

```
function MOD(a, b)
    if b ≤ 0 then
        return -1
    end if
    div = a/b
    return a - div*b
end function
```

```
def mod(a, b):
        if b <= 0:
                return -1
        div = a//b
        return a - div*b
```

All of the operations are done in constant time, and thus the algorithm runs in $O(1)$.

Problem VI.4. In pseudocode and Python:

```
function DIV(a, b)
    count = 0
    sum = 0
    while sum ≤ a do
        sum = sum + b
        count = count + 1
    end while
    return count
end function
```

```
def div(a, b):
        count = 0
        summ = 0
        while summ <= a:
                summ += b
                count += 1
        return count
```

The work performed in the while loop is constant time, and the number of loops the loop performs is $\lfloor a/b \rfloor$, and therefore the runtime is $O(\lfloor a/b \rfloor)$.

Problem VI.5. In pseudocode and Python:

```
function SQRT(n)
    return sqrt_helper(n, 1, n)
end function

function SQRT_HELPER(n, min, max)
    if max < min then
        return -1
    end if
    guess = (min + max)/2
    if guess * guess = $n$ then
        return guess
    else if guess * guess < $n$ then
```

```
        return sqrt_helper(n, guess+1, max)
    else
        return sqrt_helper(n, min, guess-1)
    end if
end function
```

```
def sqrt(n):
        return sqrt_helper(n, 1, n)

def sqrt_helper(n, minn, maxx):
        if maxx < minn:
                return -1
        guess = (minn + maxx)//2
        if guess*guess == n:
                return guess
        elif guess*guess < n:
                return sqrt_helper(n, guess + 1, maxx):
        else:
                return sqrt_helper(n, minn, guess - 1):
```

I will calculate the worst-case runtime, $T(n)$. We see that when we call sqrt_helper for a number n a constant amount of work is done for the arithmetic and if statements plus the amount of work done on a number half as big as n. This is because the next call searches a region half as large as the first call (note that this is the case for either of the cases when the guess is too big or when the guess is too small). Therefore, the following recurrence describes the worst-case run time of the algorithm: $T(n) = T(n/2) + c$ with $T(1) = O(1)$ (the base case corresponds to when the range over which to search is a single number, in which case the square root is definitely found). Solving the recurrence:

$$\begin{aligned} T_n &= T\left(\frac{n}{2}\right) + c \\ &= T\left(\frac{n}{4}\right) + c + c \\ &= T\left(\frac{n}{8}\right) + c + c + c \\ &\vdots \\ &= T\left(\frac{n}{2^k}\right) + kc. \end{aligned}$$

We can solve for the value of k which gets down to the base case by setting $n/2^k$ equal to 1 and solving for k $(= \lg n)$. Plugging the base case in:

$$\begin{aligned} T_n &= O(1) + c\lg n \\ &= T\left(\frac{n}{4}\right) + c + c \\ &= O(\lg n). \end{aligned}$$

The naive implementation of this algorithm would be to do a linear search over the range 1 to n, checking at each iteration whether that number squared is equal to n. Clearly this results in a time complexity of $\sqrt{n}$, which is worse than the algorithm above.

Problem VI.6. In pseudocode and Python:

```
function SQRT(n)
    guess = 1
    while $guess * guess \le n$ do
        if guess*guess = n then
            return guess
        end if
        guess = guess +1
    end while
    return -1
end function
```

```
def sqrt(n):
        while guess*guess <= n:
                if guess*guess == n:
                        return guess
                guess += 1
        return -1
```

Note that this is the linear scan method and as mentioned above it will have a run time of $O(\sqrt{n})$ since we find the square root by on the $\sqrt{n}$ iteration of the loop and we do constant work in each call.

Problem VI.9. In pseudocode and Python:

```
function COPYARRAY(arr)
    Let copy[·] be a new, empty array
    for $i = 0$ to $arr.length - 1$ do
        copy = appendToNew(copy, $arr[i]$)
    end for
    return copy
end function

function APPENDTONEW(arr, value)
    Let bigger[$0 \ldots arr.length$] be a new array
    for $i = 0$ to $arr.length$ do
        bigger[$i$] = $arr[i]$
    end for
    bigger[bigger.$length - 1$] = $value$
    return bigger
end function
```

```
def copyArray(arr):
        copy = []
        for val in arr:
                copy = appendToNew(copy, val)
        return copy

def appendToNew(arr, value):
        bigger = [0]*(len(arr)+1)
        for i in range(len(arr)):
                bigger[i] = arr[i]
        bigger[len(bigger)-1] = value
        return bigger
```

Problem VI.10. In pseudocode and Python:

```
function sumDigits($n$)
    $sum = 0$
    while $n > 0$ do
        $sum = sum + n$ % 10
        $n = n//10$
    end while
    return sum
end function
```

```
def sumDigits(n)
        summ=0
        while n > 0:
                summ = summ + n % 10
                n = n//10
        return summ
```

We see that in this program, the while loop results in the following sequence for n: $n, n/10, n/(10^2), \ldots n/(10^k)$, where k is the smallest integer such that $n/10^k \geq 1$. Note that k is the number of times the loop executes. Solving for k I find that $k = \lfloor \log n \rfloor = O(\lg n)$. Since there is a constant time amount of work done in each iteration of the loop, the running time is therefore $O(\lg n)$.

Problem VI.12. First sorting the array, B, takes $O(b \lg b)$ time, and then iterating through a and checking via binary search whether the element is also in B takes $O(a)O(\lg b)$ time (where the first term comes from iterating through A and the second term comes from performing a binary search in B). Thus we have the total running time of the algorithm as $O(b \lg b + a \lg b)$.