Exercises

1. Are the following statements true or not? No proof is required.

   a. $n + 1 \in \Theta(n)$

   > True

   b. $2n \in \Omega(n^2)$

   > False

   c. $n + \log n \in \mathcal{O}(n)$

   > True

   d. $2^{n+1} \in \Theta(2^n)$

   > True ($c = 2, n_0 = 0$)

   e. $n\sqrt{n} \in \mathcal{O}(n \log n)$

   > False (since $\frac{\log n}{\sqrt{n}} \to 0$ as $n \to \infty$)

   f. $n! \in \mathcal{O}(2^n)$

   > False

2. Recall that in order to prove $f \in \mathcal{O}(g)$ one has to choose $c > 0$ and n_0 and then prove that $f(n) \leq cg(n)$ for all $n \geq n_0$.

   a. Prove $n + 37 \in \mathcal{O}(n)$.

   > Take $n_0 = 2$ and $c = 37$. Then $\frac{n+37}{37n} = \frac{1}{37} + \frac{1}{n}$. For $n \geq 2, 0 \leq \frac{1}{n} \leq \frac{1}{2}$, so that $\forall n \geq n_0 : \frac{n+37}{37n} \leq \frac{1}{37} + \frac{1}{2} < 1$, so that $\forall n \geq n_0 : n + 37 \leq cn$, which was to be proven.

   b. Prove that $\frac{1}{2}n(n-1) \in \mathcal{O}(n^2)$.

   > Take $n_0 = 1$ and $c = 1$. We see that for $n \geq 1$, $\frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} - \frac{1}{2n} \leq \frac{1}{2} \leq 1$. So we conclude $\forall n \geq 0 : \frac{1}{2}n(n-1) \leq n^2$

   c. Prove that $n \in \mathcal{O}(2^n)$. (Hint: use $c = 1$ and $n_0 = 1$, then prove the inequality with induction.)

   > We use $c = 1$ and $n_0 = 1$. We will show the statement by induction:
   >
   > - Induction basis: $1 < 2 = 2^1$, which shows the statement is true for $n = 1$
   > - Induction hypothesis: suppose for some $n \geq 1$, it holds that $n \leq 2^n$.
   > - Induction step: then we show the statement for $n + 1$: $n + 1 \leq 2^n + 1$ and $1 \leq 2^n$ for $n \geq 1$, so $n + 1 \leq 2^n + 2^n = 2 \cdot 2^n = 2^{n+1}$, which completes the proof.

```
search(int v[], int n) {
    int i = 0;
    bool foo = false;
    while (i < n && !foo) {
        if (v[i] > 42) {
            foo = true;
        }
    i++;
    }
    return foo
}
```

3. Consider the above algorithm.

    a. What is the worst case scenario? How many operations does it take in this case (your answer should use $\mathcal{O}$ notation and depend on n)?

> We write the largest possible number of repetitions and the cost on every line:

```
search(int v[], int n) {
    int i = 0;              //c_1, repetitions: 1
    bool foo = false;       //c_2, repetitions: 1
    while (i < n && !foo) { //c_3, repetitions: n
        if (v[i] > 42) {    //c_4, repetitions: n
            foo = true;     //c_5, repetitions: n
        }
    i++;                    //c_6, repetitions: n
    }
    return foo              //c_7, repetitions: 1
}
```

> We conclude that the runtime cost is $T(n) = (c_1 + c_2 + c_7) + (c_3 + c_4 + c_5 + c_6)$, so it is $\mathcal{O}(n)$.

    b. What is the best case scenario? How many operations does it take in this case (your answer should use $\mathcal{O}$ notation and depend on n)?

> In the best case, the condition `if (v[i] > 42)` is always `false` so that line 5 is never executed: we get a cost of $T(n) = (c_1 + c_2 + c_7) + (c_3 + c_4 + c_6)$, so it is $\mathcal{O}(n)$. It is still linear, so ignoring constants there is no difference (note that the function is not written so that it can terminate early, which would be admitted by its semantics).

4. Consider the following algorithm, which takes as input an integer array v of length n. To analyse this algorithm, we will only count array access operations on line 4.

    a. Show that line 4 is a good measure of the complexity of the algorithm, by showing that the number of times each other line is performed on a given input is in `\mathcal O(#4)`, where `#4` is the number of times line 4 is executed.

line 5 up to 11 are all constant-time statements, that only follow after line 4. If we know how often line 4 is executed, then this gives an upper bound for how often lines behind line 4 (up to line 11) are executed. So that makes the algorithm `\mathcal O(#4)`, since the other lines 1,2, 12, 13 have either constant-time operations which are dominated by `\mathcal O(#4)` anyway, or are not actually statements but are simply part of the syntax of our pseudo-language.

b. How many times is Line 4 executed in the worst case (your solution should depend on n)? Describe some inputs for which this worst case is achieved.

In the worst case, we only increment `i` or decrement `j` once per iteration, meaning that they move closer to each other only `1` per execution of line 4. That means that line 4 is executed for $i = 0, \dots, n-1$ if j is not decreased, for example. It does not matter how often or in what order `i` and `j` are decreased, if only one of them is decreased/increased per iteration, we get `#4 = n`.

Some inputs would be: `v = [2,2,2,2,2], n = 5`. Here all entries of `v` are even, so we always increase `i` by 1 every iteration. Another would be `v = [3,3,3], n = 3`. Here we decrease `j` only by `1` every step, making `#4 = n` effectively.

c. How many times is Line 4 executed in the best case (your solution should depend on n)? Describe some inputs for which this best case is achieved.

In the best case, we always have i even and j odd, in which case we are allowed to both increase i and decrease j, making i and j move together `2` per iteration. In that case `#4 = \lfloor \frac n2 \rfloor`, the floor of $\frac{n}{2}$.

Example input: `v = [3,3,2], n = 3`. We swap `v[0]` with `v[2]` and make `i = 1 j = 1` for one execution of line 4, and after that the condition in the `while` fails and we exit the procedure. Another example input: `v = [3,5,2,2], n = 4`: We iterate twice and then `v = [2,2,3,5], i = 2, j = 1` and we only executed line 4 $2 = \frac{4}{2}$ times.

d. Compare the best and worst case from your previous two solutions asymptotically, that is, using the $\mathcal{O}\Theta, \Omega$ notation.

There is no difference asymptotically, because we can supress the constant costs of all branches, and thus the complexity of the algorithm is really `\Theta(#4)`, but $\lfloor \frac{n}{2} \rfloor \in \Theta(n)$ so in both best and worst case the runtime complexity is $\Theta(n)$ or completely equivalently $\Theta(\lfloor \frac{n}{2} \rfloor)$.

e. Suppose we make a mistake and forget the assignment `j--` on line 9. Does the algorithm still work? How does this affect the best case? How does this compare asymptotically to the original best case?

Given that the semantics of the original algorithm would be to (by swapping) place all even elements in the initial segment of the array and place all odd elements in the tail segment of the array, this would not change the behaviour of the algorithm, because `j` is decreased anyway if it is odd, but only in the next iteration. So in the best case, we are not allowed to decrease `j` in the same step, meaning that it takes n iterations instead of $\lfloor \frac{n}{2} \rfloor$ in the best case now.

5. The n-th harmonic number is the sum of the reciprocals of the first n natural numbers such that $H_n = \sum_{k=1}^{n} \frac{1}{k}$. Write a function returning H_n with $n \in \mathbb{N}$ and comment on its asymptotic time complexity.

> The following solution, supposing that a rational number can be respresented as a pair of `int`s, namely `{numerator, denominator}`. We then use that $\frac{l}{T} + \frac{1}{k} = \frac{kl+T}{kT}$ and ignore overflows or simplifications using the Greatest Common Divisor.

```
struct rational {
    int num;
    int den;
};

{int, int} harmonic (int n) {
    // Assumes that n >= 1.
    rational temp {1,1};            //c, 1 repetition
    for (int k = 1; k <= n; k++) {  //c_0, n repetitions
        temp.num *= k;              //c_1, n repetitions
        temp.num += temp.den;       //c_2, n repetitions
        temp.den *= k;              //c_3, n repetitions
    }
    return temp;                    //b, 1 repetitions
}
```

> This algorithm has in all cases (best, average, worst) the same runtime complexity $T(n) = (c_0 + c_1 + c_2 + c_3)n + (b + c) \in \Theta(n)$. It is exact but does not simplify the fraction. I hope that you indeed looked for exact arithmetic.