Exercise 1: Maximum Profit

You are given a vector of daily stock price movements for a company. It contains the relative changes in the stock price for each day, for a total of N consecutive trading days. You want to know the maximum profit you could have made by buying the stock at the start of one day and selling it at the start of another. (You can only buy and sell once, but have the option to not buy and sell at all. You can sell after the last day.)

Example: For example, with price changes $[1.1, 0.7, 1.2, 1.2, 0.5]$: if the stock price rose 10% on day 1, buying on day 1 and selling on day 2 yields a profit factor of 1.1. Buying on day 3 and selling on day 5 gives a profit factor of $1.2 \cdot 1.2 = 1.44$, which is the maximum achievable profit.

You come up with the following algorithm:

```r
getMaxProfit <- function(price.changes) {
  checkmate::assertNumeric(price.changes, lower = 0,
    any.missing = FALSE, finite = TRUE, min.len = 1)
  n <- length(price.changes)
  max.profit <- 1.0
  for (day.buy in seq_len(n)) {
    # loop to n + 1 to sell after the last day
    for (day.sell in seq(day.buy + 1, n + 1)) {
      profit <- prod(price.changes[seq(day.buy, day.sell - 1)])
      if (profit > max.profit) {
        max.profit <- profit
      }
    }
  }
  max.profit
}

# example:
getMaxProfit(c(1.1, 0.7, 1.2, 1.2, 0.5))

#> [1] 1.44
```

(a) What is the asymptotic time complexity of this algorithm, in terms of the length of the input vector?

(b) Come up with a simple change to the algorithm that makes it asymptotically faster.

(c) *Bonus:* Come up with an algorithm that has time complexity $\Theta(N)$.

Exercise 2: Search Complexity

The following functions search for an element in a sorted vector. They are given a sorted vector `x` and a "search key" `key` and return `TRUE` if `key` is found in `x` and `FALSE` otherwise. For each of these, you should give the worst-case asymptotic time complexity in terms of the size N of the vector `x`, expressed as $\Theta(f(N))$.

(a) *Binary search* is a classic algorithm that searches for an element in a sorted vector. It works by checking whether the middle element of the vector is less than, equal to, or greater than the search key. If it is equal, the function returns `TRUE`. If it is less than the search key, the function knows that the search key must be in the second half of the vector. It then calls itself recursively with just the second half of the vector. Similarly with the first half of the vector if the search key is less than the middle element. To avoid copying the vector, we call an inner function with just the indices of the vector to consider.

```
# search for `key` in sorted vector `x`
binarySearch = function(x, key) {
  checkmate::assertNumeric(x)
  checkmate::assertNumber(key)

  # search for `key` in `x[left:right]`
  binarySearchInner = function(x, key, left, right) {
    if (left > right) {
      return(FALSE)
    }
    mid = (left + right) %/% 2
    mid.val = x[[mid]]
    if (mid.val == key) {
      # found the key
      return(TRUE)
    } else if (mid.val < key) {
      # search in the right half, since `key` is greater than `mid.val`
      return(binarySearchInner(x, key, mid + 1, right))
    } else {
      # search in the left half
      return(binarySearchInner(x, key, left, mid - 1))
    }
  }
  binarySearchInner(x, key, 1, length(x))
}
```

What is the worst-case asymptotic time complexity of this algorithm?

(b) Trying to be original, we come up with an algorithm we call "ternary search". It works by taking items at 1/3 and 2/3 of the way through the vector and checking whether the search key is less than, equal to, or greater than these items. It then calls itself recursively with just the third of the vector that the search key must be in.

```
# search for `key` in sorted vector `x`
ternarySearch = function(x, key) {
  checkmate::assertNumeric(x)
  checkmate::assertNumber(key)

  # search for `key` in `x[left:right]`
  ternarySearchInner = function(x, key, left, right) {
    if (left > right) {
      return(FALSE)
    }
    third = (right - left) %/% 3
    mid1 = left + third  # point 1/3 of the way through the vector
    mid2 = left + 2 * third  # point 2/3 of the way through the vector
    mid1.val = x[[mid1]]
    mid2.val = x[[mid2]]
    if (mid1.val == key || mid2.val == key) {
      return(TRUE)
    } else if (mid1.val > key) {
      # search in the first third of the vector
      return(ternarySearchInner(x, key, left, mid1 - 1))
    } else if (mid2.val > key) {
      # search in the second third of the vector
      return(ternarySearchInner(x, key, mid1 + 1, mid2 - 1))
    } else {
      # search in the last third of the vector
      return(ternarySearchInner(x, key, mid2 + 1, right))
    }
```

```
  }
  ternarySearchInner(x, key, 1, length(x))
}
```

What is the worst-case asymptotic time complexity of this algorithm?

(c) We consider the case where we may expect the search key to often be near the beginning of the vector. For this, we write the following variant, which we call "skewed binary search". It works like binary search, but uses the point at the first $1/10^{\text{th}}$ of the way through the vector as the pivot. Note that, besides the variable name change, the only difference to (a) is that the calculation of the pivot uses `%/% 10` instead of `%/% 2`.

```
# search for `key` in sorted vector `x`
skewedBinarySearch = function(x, key) {
  checkmate::assertNumeric(x)
  checkmate::assertNumber(key)

  # search for `key` in `x[left:right]`
  skewedBinarySearchInner = function(x, key, left, right) {
    if (left > right) {
      return(FALSE)
    }
    pivot = left + (right - left) %/% 10
    pivot.val = x[[pivot]]
    if (pivot.val == key) {
      return(TRUE)
    } else if (pivot.val < key) {
      return(skewedBinarySearchInner(x, key, pivot + 1, right))
    } else {
      return(skewedBinarySearchInner(x, key, left, pivot - 1))
    }
  }
  skewedBinarySearchInner(x, key, 1, length(x))
}
```

What is the worst-case asymptotic time complexity of this algorithm?

(d) We want to skew the search even more. For our algorithm "extra skewed binary search", we use the point at the first $1/10^{\text{th}}$ of the way through the vector as the pivot, but limit the search to at most 10 items to the right of the current beginning of the vector under consideration. Note the only change that was made here is the `min()` call in the calculation of the pivot.

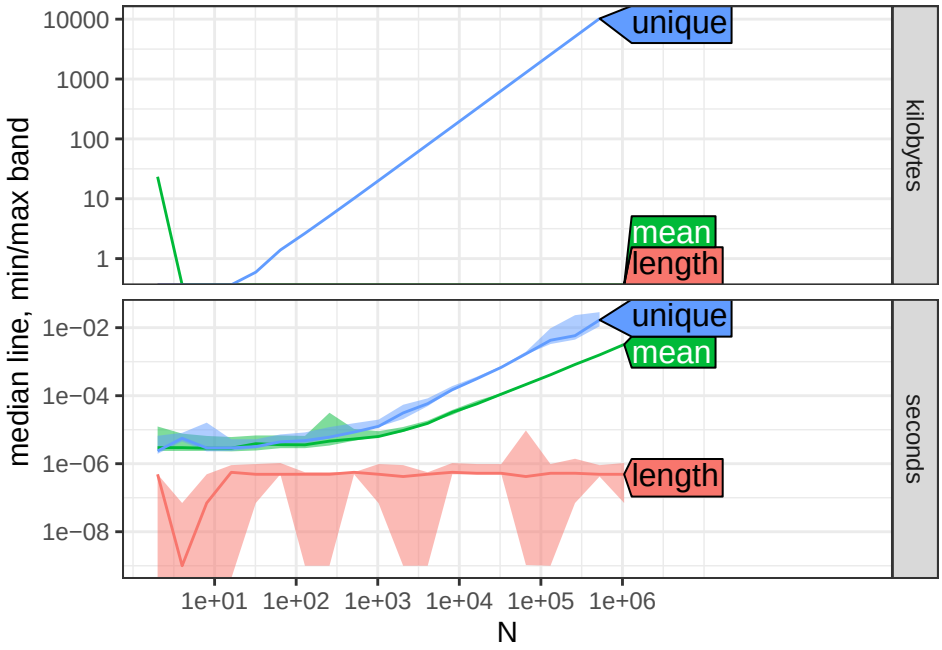
```
extraSkewedBinarySearch = function(x, key) {
  checkmate::assertNumeric(x)
  checkmate::assertNumber(key)

  extraSkewedBinarySearchInner = function(x, key, left, right) {
    if (left > right) {
      return(FALSE)
    }
    pivot = left + min((right - left) %/% 10, 10)
    pivot.val = x[[pivot]]
    if (pivot.val == key) {
      return(TRUE)
    } else if (pivot.val < key) {
      return(extraSkewedBinarySearchInner(x, key, pivot + 1, min(right, pivot + 10)))
    } else {
      return(extraSkewedBinarySearchInner(x, key, left, pivot - 1))
    }
  }
  extraSkewedBinarySearchInner(x, key, 1, length(x))
}
```

What is the worst-case asymptotic time complexity of this algorithm?

Exercise 3: `atime`

The `atime` package can be used to benchmark "asymptotic" performance of R expressions:

```
result <- atime::atime(
  setup = {
    x <- runif(N)
  },
  length = length(x),
  mean = mean(x),
  unique = unique(x)
)
plot(result)
```



(a) Use the `atime` package to benchmark the various sorting methods built into R: `sort(x, method = <method>)`, for `<method>` in `"radix"`, `"shell"`, and `"quick"`. Run the benchmark for the following cases:

(i) A vector of N distinct randomly ordered numbers.

(ii) The vector `1:N`.

(iii) A vector of N distinct numbers that are already sorted in increasing order, but which is *not* `1:N`. Do not use `sort()` to generate this input vector.

What do you conclude from the results? What is the "best" sorting method?

(b) Use `atime::atime()` in combination with `atime::references_best()` to estimate the asymptotic complexity of the following functions, applied to a vector `runif(N)`:

(i) `mean(x)`

(ii) `length(x)`

(iii) `unique(x)`

(iv) `any(x > 0)`

(v) `all(x > 0)`

(vi) `sort(x)`

(vii) `x[-1]`

(viii) `x[-length(x)]`

Do the results match your expectations? Are any of the results surprising?