Solution 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 $\Theta(N)$.

(a) What is the asymptotic time complexity of this algorithm?

The algorithm `getMaxProfit` has two explicit nested loops. The outer loop for `day.buy` runs N times, where N is `length(price.changes)`. The inner loop for `day.sell` runs $\Theta(N)$ times on average, for each outer loop iteration. Inside the inner loop, the operation `prod(price.changes[seq(day.buy, day.sell - 1)])` calculates the product of a sequence of numbers. The length of this sequence, `day.sell - day.buy`, can be up to $\Theta(N)$. Computing the product of k numbers takes $\Theta(k)$ time. Therefore, the total time complexity is $\Theta(N \cdot N \cdot N) = \Theta(N^3)$.

(b) Come up with an algorithm that is asymptotically faster.

We can improve the calculation of the product. Instead of recomputing the product from scratch in the innermost part of the original algorithm, we can maintain a running product for a fixed `day.buy` as `day.sell` increases. This eliminates one order of N.

```r
getMaxProfitN2 <- 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)) {
    current.product <- 1
    for (day.sell in seq(day.buy + 1, n + 1)) {
      current.product <- current.product * price.changes[day.sell - 1]
      if (current.product > max.profit) {
        max.profit <- current.product
      }
    }
  }
  max.profit
}
```

This algorithm has two nested loops. The outer loop (indexed by `day.buy`) runs N times. The inner loop (indexed by `day.sell`) runs up to $\Theta(N)$ times. Inside the inner loop, operations are $\Theta(1)$ (multiplication and comparison). Thus, the complexity is $\Theta(N^2)$.

Alternatively, we can pre-compute the cumulative product and index into it. It needs $\Theta(N)$ more space.

Note: Generally, `cumprod` can become very small or very large. Depending on the input your algorithm expects, it is sometimes better to use `cumsum` on logarithms. This is not the point of this exercise here, however.

```r
getMaxProfitCumprodN2 <- 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
  # the cumulative profit represents the profit if we buy at day 1 and sell at day i
  cum.profit <- c(1, cumprod(price.changes))

  for (day.buy in seq_len(n)) {
    for (day.sell in seq(day.buy + 1, n + 1)) {
      current.product <- cum.profit[day.sell] / cum.profit[day.buy]
      if (current.product > max.profit) {
        max.profit <- current.product
      }
    }
  }
  max.profit
}
```

(c) *Bonus:* Come up with an algorithm that has time complexity $\Theta(N)$. This problem is a variation of the maximum sum (when taking the logarithm) subarray problem[1], for which *Kadane's algorithm* is a well-known $O(N)$ solution.

```r
getMaxProfitN <- function(price.changes) {
  checkmate::assertNumeric(price.changes, lower = 0,
    any.missing = FALSE, finite = TRUE, min.len = 1)
  n <- length(price.changes)
  best.profit.so.far <- 1.0
  best.profit.overall <- 1.0

  # writing '+1' for consistency with the other algorithms, but would not be
  # necessary here
```

[1] https://en.wikipedia.org/wiki/Maximum_subarray_problem

```r
  for (day.sell in seq_len(n) + 1) {
    last.day.profit <- price.changes[[day.sell - 1]]
    # 'best.profit.so.far' holds the best profit I could get if I buy on any previous
    # day and hold until the day before 'day.sell'.
    # Now the best profit I can get if I hold until 'day.sell' is either that
    # times the profit of the last day, or *only* the profit of the last day
    # (if holding until the day before 'day.sell' would have resulted in a loss).
    best.profit.so.far <- max(best.profit.so.far * last.day.profit, last.day.profit)

    # If 'best.profit.overall' is surpassed, we update it.
    best.profit.overall <- max(best.profit.overall, best.profit.so.far)
  }
  best.profit.overall
}
```

This algorithm iterates through the input vector once, performing a constant number of operations (multiplications, comparisons, assignments) at each step. Therefore, its time complexity is $\Theta(N)$.

Solution 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.

```r
# 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?

(a) **Binary Search**

The worst-case asymptotic time complexity of binary search is $\Theta(\log N)$.

In each step of the `binarySearchInner` function, the size of the search interval is approximately halved. The work done in each step (comparisons, arithmetic operations, recursive call setup) is constant, $\Theta(1)$. The recurrence relation for the worst-case time complexity $T(N)$ for an initial vector of size N is therefore $T(N) = T(N/2) + \Theta(1)$. This is a standard recurrence relation that solves to $T(N) \in \Theta(\log N)$. To see this, consider the following: Let N be the initial size of the vector. After 1 step, the size is (approximately) $N/2$. After 2 steps, it's $N/4$. After k steps, the size is $N/2^k$. The algorithm stops when the problem size is reduced to a constant (e.g., 1 or 0). So we need to find k such that $N/2^k \approx 1$. This gives $2^k \approx N$, or $k \approx \log_2 N$. Since each of these k steps takes $\Theta(1)$ time, the total time is $k \times \Theta(1) = \Theta(\log N)$. The worst case typically occurs when the element is not found, or found only after the search space has been reduced to a single element, thus requiring the maximum number of divisions.

More formally, by induction: Let $T(N)$ be the time complexity. We want to show $T(N) \leq c \log N$ for some constant c and sufficiently large N. Base case: Consider $N = 2$ and $N = 3$, for which $T(N)$ is at most a constant, say k_0. We can choose c large enough such that $k_0 \leq c \log N$ for $N \in \{2, 3\}$. Inductive hypothesis (IH): Assume $T(M) \leq c \log M$ for all $2 \leq M < N$ (where M must be required to be large enough for the base case). Inductive step for $N > 3$: We have $T(N) = T(\lfloor N/2 \rfloor) + k_1$ for some constant k_1 representing $\Theta(1)$ work. By IH, $T(\lfloor N/2 \rfloor) \leq c \log(\lfloor N/2 \rfloor) \leq c \log(N/2)$, since $\lfloor N/2 \rfloor \leq N/2$ for positive N. Then $T(N) \leq c \log(N/2) + k_1 = c(\log N - \log 2) + k_1 = c \log N - c \log 2 + k_1$. For this to be $\leq c \log N$, we need $-c \log 2 + k_1 \leq 0$, which means $k_1 \leq c \log 2$. Since k_1 and $\log 2$ are positive constants, we can always choose a c (e.g., $c \geq k_1 / \log 2$) large enough to satisfy this condition and the base cases. Thus, $T(N) \in O(\log N)$. A similar argument (using $T(N) \geq c' \log N$ and $\lfloor N/2 \rfloor \geq (N-1)/2$) can show $T(N) \in \Omega(\log N)$, so $T(N) \in \Theta(\log N)$.

(b) **Ternary Search**

The worst-case asymptotic time complexity of ternary search is $\Theta(\log N)$.

In each step, the search interval is divided into three parts, and the algorithm recurses on one of these thirds. The work done in each step is constant, $\Theta(1)$. (The behavior for $N < 3$ where `third` is rounded down to 0 involves a few constant-time steps and doesn't change the overall asymptotic complexity for large N.) The recurrence relation for the worst-case time complexity $T(N)$ is $T(N) = T(\lceil N/3 \rceil) + \Theta(1)$, which is also solved by $T(N) \in \Theta(\log N)$: Let N be the initial size. After 1 step, the size is $\lceil N/3 \rceil$. After 2 steps, it's $\lceil N/9 \rceil$. After k steps, the size is $N/3^k$. The algorithm stops when $N/3^k \approx 1$, which means $3^k \approx N$, or $k \approx \log_3 N$. Since each of these k steps takes $\Theta(1)$ time, the total time is $k \times \Theta(1) = \Theta(\log N)$. The base of the logarithm (2 for binary search, 3 for ternary search) changes, but this does not affect the Θ-complexity class, as logarithms of different constant bases differ only by a constant factor (e.g., $\log_3 N = \log_2 N / \log_2 3$).

A more formal proof by induction looks similar to the proof for binary search.

Looking at this naively, it may seem like ternary search is faster than binary search by a constant factor, since the logarithm has a larger base. However, ternary search has a larger "$\Theta(1)$" in each recursive call (it compares 2 elements instead of 1 to the key), so the constant factor for ternary search is in fact larger.

(c) **Skewed Binary Search**

The worst-case asymptotic time complexity of skewed binary search is $\Theta(\log N)$.

The pivot is chosen at the $1/10^{\text{th}}$ position. In the worst case, the search continues in the larger segment, which is 9/10 of the current interval size. The work done in each step is constant, $\Theta(1)$. The recurrence relation for the worst-case time complexity $T(N)$ is $T(N) = T(9N/10) + \Theta(1)$. This recurrence relation solves to $T(N) \in \Theta(\log N)$: Let N be the initial size. After 1 step, the size is $(9/10)N$. After 2 steps, it's $(9/10)^2 N$. After k steps, the size is $(9/10)^k N$. The algorithm stops when $(9/10)^k N \approx 1$, which means $(10/9)^k \approx N$, or $k \approx \log_{10/9} N$. Since each of these k steps takes $\Theta(1)$ time, the total time is $k \times \Theta(1) = \Theta(\log N)$. The constant factor in the logarithm will be larger than for standard binary search because the problem size reduces more slowly, but the asymptotic growth rate remains logarithmic.

A more formal proof by induction looks similar to the proof for binary search.

(d) **Extra Skewed Binary Search**

The worst-case asymptotic time complexity of extra skewed binary search is $\Theta(N)$.

In this algorithm, the pivot is calculated as `pivot = left + min((right - left) %/% 10, 10)`. Consider the worst-case scenario where `pivot.val < key` in every step. Let $M = \text{right} - \text{left} + 1$ be the size of the current search interval. The current `left` pointer advances by $c =$ `min((M-1) %/% 10, 10) + 1` positions. As established:

- If $M \geq 101$, `left` advances by $c = 11$ positions.
- If $M < 101$, `left` advances by c positions where $1 \leq c \leq 10$.

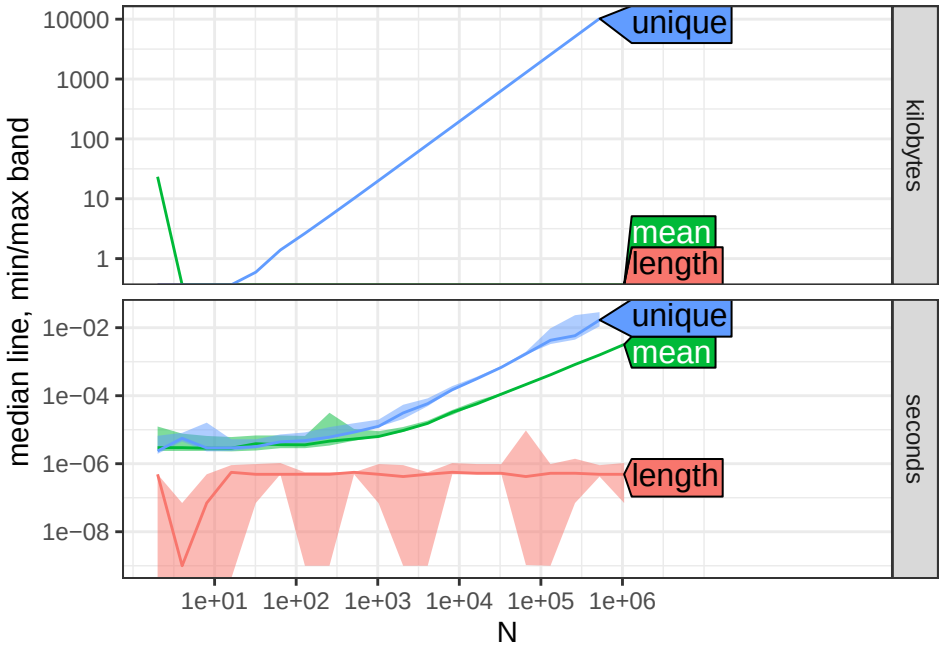
In every recursive step, which performs $\Theta(1)$ work, the `left` pointer advances by at least 1 position and at most 11 positions. This means the effective size of the problem (the remaining range for `left` to cover) is reduced by a small constant amount c in each step, rather than by a multiplicative factor. If the initial interval has N elements, and in each step we reduce the number of elements to consider by at least 1 (and at most 11), it will take at most N steps (and at least $N/11$ steps) to reduce the problem to a trivial size. Since each step costs $\Theta(1)$, the total time is $N \times \Theta(1) = \Theta(N)$ in the worst case. The algorithm essentially performs a linear scan, although it might skip a small constant number of elements at each step.

Solution 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]`

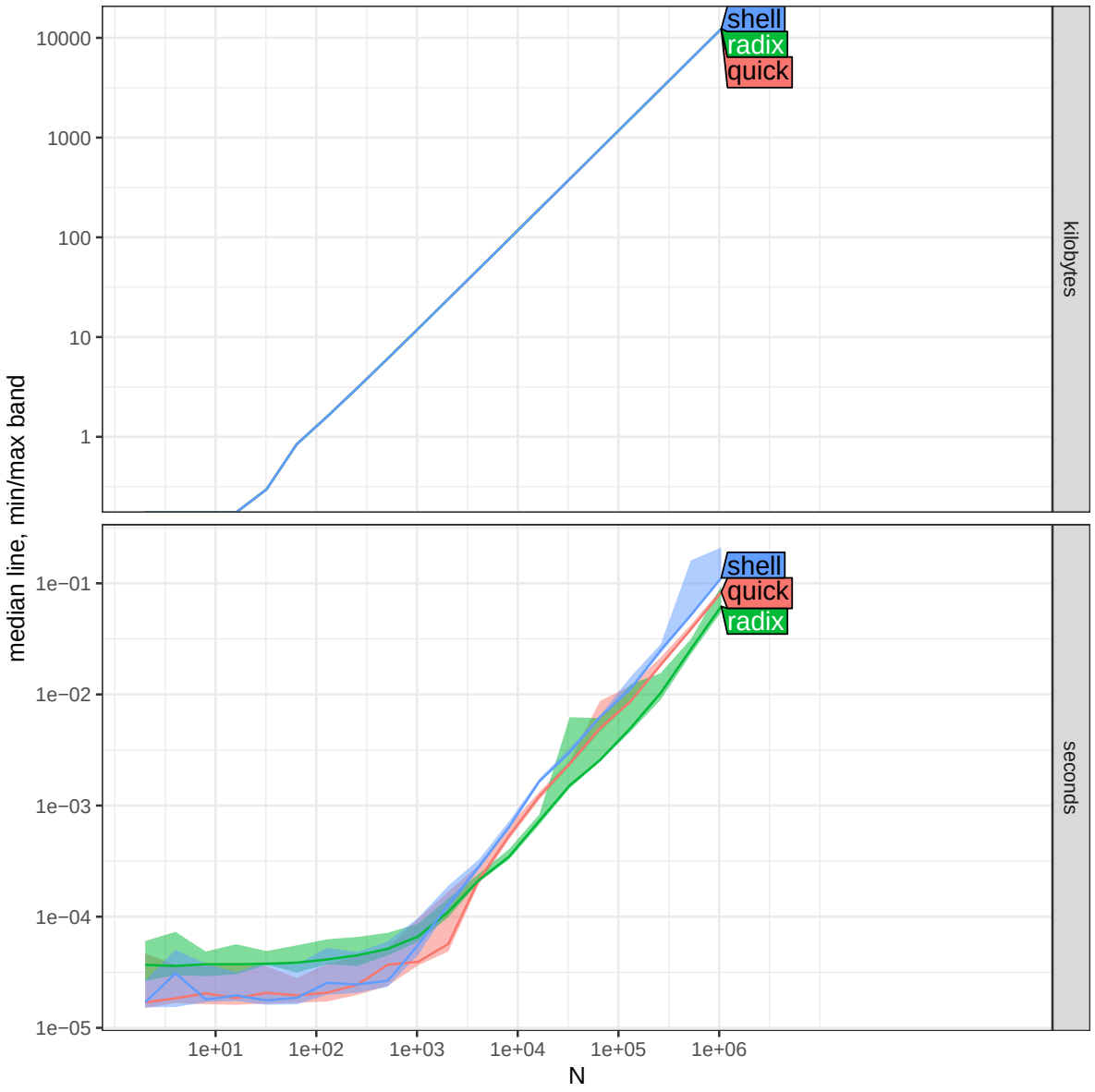
(viii) `x[-length(x)]`

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

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

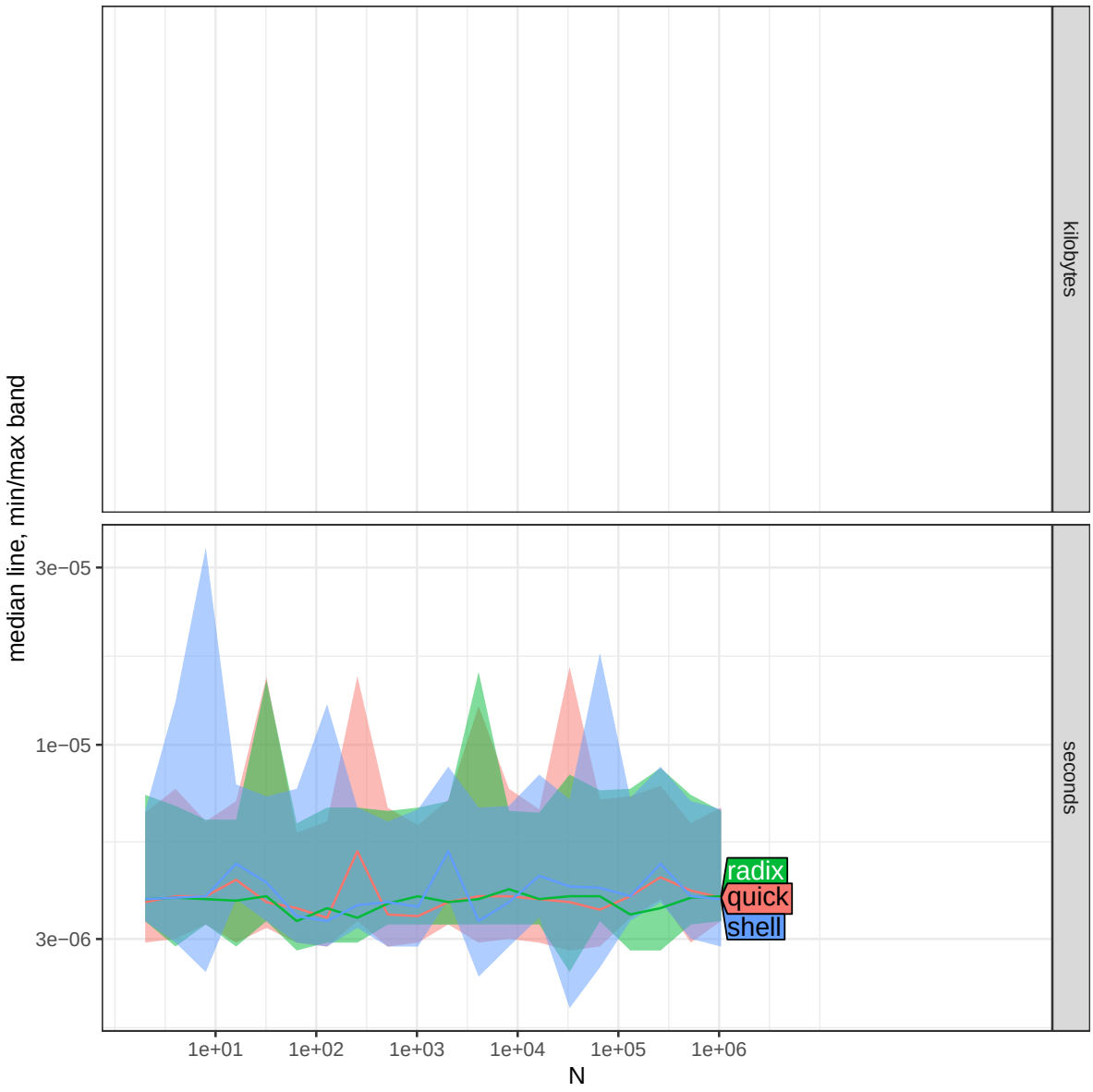
```
result <- atime::atime(
  seconds.limit = 1,
  times = 30,
  setup = {
    x <- runif(N)
```

```
  },
  radix = sort(x, method = "radix"),
  shell = sort(x, method = "shell"),
  quick = sort(x, method = "quick")
)
plot(result)
```



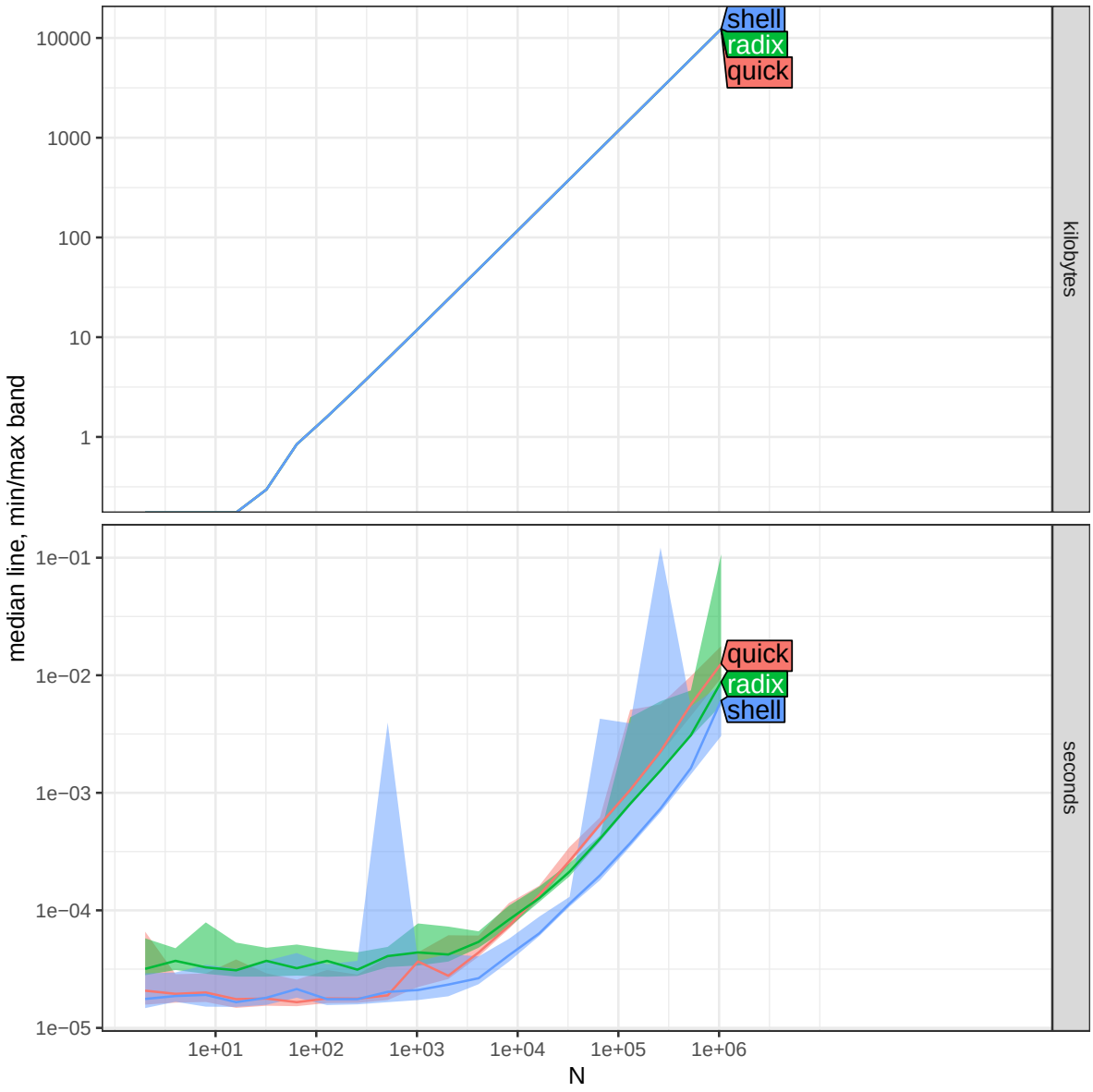
(ii) The vector `1:N`.

```
result <- atime::atime(
  seconds.limit = 1,
  times = 30,
  setup = {
    x <- 1:N
  },
  radix = sort(x, method = "radix"),
  shell = sort(x, method = "shell"),
  quick = sort(x, method = "quick")
)
plot(result)
```



(iii) A vector of N distinct numbers that are already sorted in increasing order, but which is *not* `1:N`.

```
result <- atime::atime(
  seconds.limit = 1,
  times = 30,
  setup = {
    x <- cumsum(runif(N))
  },
  radix = sort(x, method = "radix"),
  shell = sort(x, method = "shell"),
  quick = sort(x, method = "quick")
)
plot(result)
```

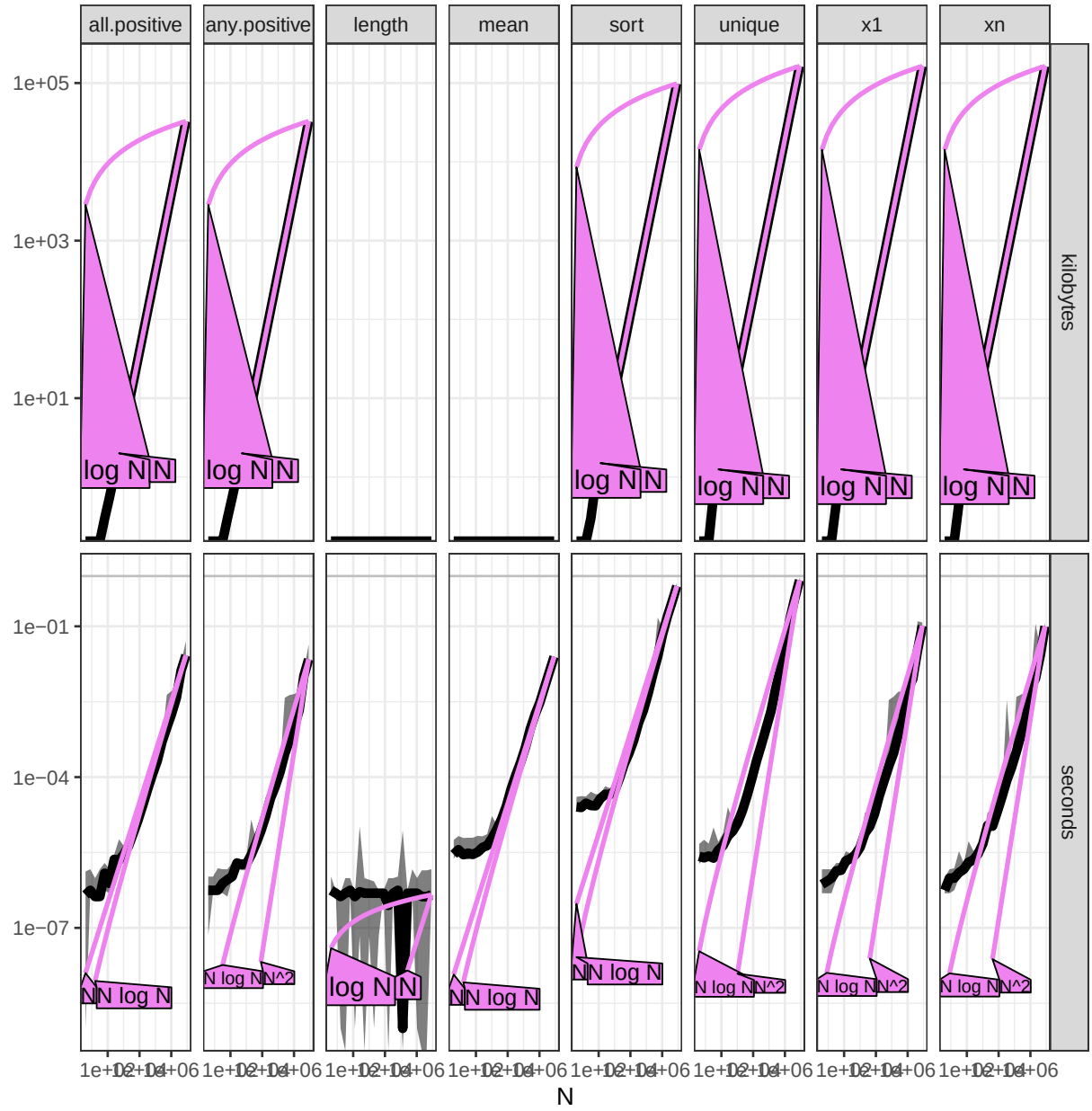


Observations:

- There is no "best" sorting method, different methods can be slightly faster for different cases.
- For short vectors, around $N \leq 1000$, the time complexity is constant; the `"radix"` method is slower than `"shell"` and `"quick"`.
- For long vectors of random numbers, `"radix"` works best.
- For vectors that are already (mostly) sorted, `"shell"` does well.
- In any case, sorting vectors that are (mostly) sorted is faster by a factor of around 10.
- R recognizes that `1:N` is already sorted and returns in constant time.

(b)
```
result <- atime::atime(
  seconds.limit = 1,
  times = 10,
  N = 2^seq(2, 23),
  setup = {
    x <- runif(N)
  },
  mean = mean(x),
  length = length(x),
```

```
  unique = unique(x),
  any.positive = any(x > 0),
  all.positive = all(x > 0),
  sort = sort(x),
  x1 = x[-1],
  xn = x[-length(x)]
)
rbest <- atime::references_best(result)
plot(rbest)
```



Observations:

- `length(x)` is $O(1)$: the length of the vector is saved in the vector's metadata.
- `mean(x)` is $O(N)$: to compute the mean, all elements of the vector are accessed.
- `sort(x)` is $O(N \log N)$
- `x[-1]` and `x[-length(x)]` are both $O(N)$, since new copies of the vector are created.
- `unique(x)` should be $O(N)$, probably because of caching behavior, it looks more steep than that.
- both `any(x > 0)` and `all(x > 0)` are $O(N)$, since they check every element of the vector. This may

be surprising, since one might expect `any(x > 0)` to be $O(1)$, as all inputs are positive. However, the function `any()` is only applied once the `x > 0` was already computed for every element of `x`.

- In general, the quantitative results are relatively noisy. Inferring asymptotic behavior is made more difficult by the fact that timing is influenced by how efficiently the CPU's cache is used. When the size of data structures involved crosses cache sizes, running time increases, which can bend the run time curves more than under ideal conditions. We see that the "unit cost Random Access Machine" model is not entirely accurate.