STA 6366, Homework.4

Carson Slater *Baylor University*

1) **Suppose we are comparing 2 DNA sequences of length $N = 250$ using an exact approach (no indels, no mismatches allowed). We will construct a partial mass function for the longest matching subsequence ($Y_{(n)}$) using the Markov chain approach, given that the strings are generated independently of one another with each nucleotide having a 25% chance of appearing at any given position.**

a. **Write a function that constructs a transition matrix of length k to be used to calculate $P(Y_{(n)} \geq k)$. Note that this should depend only on k.**

```
# ------------------- Problem 1a -------------------
gen_transition_matrix <- function(k) {
  C <- diag(0.25, k)
  B <- matrix(0.75, nrow = k, ncol = 1)

  cbind(B, C) |>
    rbind(c(rep(0, k), 1))
}

gen_transition_matrix(3)
```

```
##      [,1] [,2] [,3] [,4]
## [1,] 0.75 0.25 0.00 0.00
## [2,] 0.75 0.00 0.25 0.00
## [3,] 0.75 0.00 0.00 0.25
## [4,] 0.00 0.00 0.00 1.00
```

b. **Use your function in (a) to determine $P(Y_{(n)} = k)$ for $k = 0, 1, ..., 10$ and $P(Y_{(n)} \geq 11)$.**

See Table 1.

2) **Given the same setup in (1), we will now explore a solution to this (and a related problem) via simulation.**

a. **Suppose X follows a truncated geometric distribution with probability of success p and maximum value n. What is mass function of X for a general n and p?**

Recall that the geometric distribution gives the number of failures before the first success, where each trial succeeds with probability p. A truncated geometric with maximum value n restricts the support to $\{0, 1, \ldots, n\}$, renormalizing so the probabilities sum to 1.

Table 1: Partial Mass Function for the Longest Matching Subsequence ($Y_{(n)}$)

k	$P(Y_{(n)} = k)$
0	0.000000
1	0.000002
2	0.047999
3	0.431855
4	0.354460
5	0.121706
6	0.032852
7	0.008345
8	0.002088
9	0.000520
10	0.000130
≥ 11	0.000043

For the standard geometric, $P(X = k) = (1-p)^k p$. Truncating and renormalizing gives:

$$\begin{aligned} P(X = k) &= \frac{(1-p)^k p}{\sum_{j=0}^{n}(1-p)^j p} \\ &= \frac{(1-p)^k p}{p \cdot \frac{1-(1-p)^{n+1}}{1-(1-p)}} \\ &= \frac{(1-p)^k p}{p \cdot \frac{1-(1-p)^{n+1}}{p}} \\ &= \frac{(1-p)^k p}{1-(1-p)^{n+1}} \end{aligned}$$

for $k = 0, 1, \ldots, n$, and $P(X = k) = 0$ otherwise.

b. **Use the result from (a) to simulate the distribution of $Y_{(n)}$ (the length of the longest matching subsequence in a string of 250) with at least 100k repetitions. Estimate the probabilities from 1(b).**

See Table 2.

c. **Now, we will allow for up to 2 mismatches as we hunt for the longest sequence. Tweak your simulation to account for this and find the new probabilities. Note that k should not include the mismatches (So if the total length of the sequence is 12 but it contains 2 mismatches, then $k = 10$). Hint: Do not use the negative binomial here!**

Table 2: Estimated PMF of $Y_{(n)}$: 0 vs. 2 Allowed Mismatches

k	0 Mismatches	2 Mismatches
0	0.00000	0.00000
1	0.00000	0.00000
2	0.04709	0.00078
3	0.43364	0.08617
4	0.35178	0.35602
5	0.12240	0.32551
6	0.03342	0.15242
7	0.00857	0.05379
8	0.00226	0.01725
9	0.00062	0.00567
10	0.00017	0.00169
≥ 11	0.00005	0.00070

3) **Consider the "Making Change" Problem in section 4.3 of the slides. The problem was presented using U.S. denominations (1, 5, 10, 25, 50, 100) and Roman republic denominations (120, 40, 30, 24, 20, 10, 5, 4, 1).**

a. **Write a function that implements the dynamic programming solution referenced in the slides for the Roman republic denominations.**

```r
# ------------------- Problem 3a -------------------
mnc <- function(money, denominations = c(1, 4, 5, 10, 20, 24, 30, 40, 120)) {
  # dp[i+1] stores min coins for amount i (R is 1-indexed, so amount i -> ind
  # Initialize with Inf (impossible), except dp[1] = 0 (amount 0 needs 0 coins
  dp <- rep(Inf, money + 1)
  dp[1] <- 0 # dp[amount + 1], so amount 0 -> index 1

  # Build up from small values to money (bottom-up)
  for (amount in 1:money) {
    for (coin in denominations) {
      if (coin <= amount && dp[amount - coin + 1] != Inf) {
        dp[amount + 1] <- min(dp[amount + 1], dp[amount - coin + 1] + 1)
      }
    }
  }

  result <- dp[money + 1]
  return(ifelse(result == Inf, -1, result))
}
```

b. **Verify that your function works by checking MinNumCoins(72) (output should be 3) and MinNumCoins(168) (output should be 3).**

```
# ------------------- Problem 3b -------------------
mnc(72)
```

```
## [1] 3
```

```
mnc(168)
```

```
## [1] 3
```

4) **Suppose we have two sequences x = GTTCCAGGTA and y = CGAGTAGTCGT. Use dynamic programming to solve for the optimal alignments under the below scoring schemes. Make sure that your answer includes both the optimal alignment(s) and the corresponding score.**

a. **Find the optimal alignment using a score that assigns +1 for a match, -0.5 for a mismatch and -1 for an indel.**

```
## Max alignment score: -1.5
## Found 9 optimal alignment(s):
##
## --- Alignment 1 ---
## Seq1: -G-TTCCAGGTA
## Seq2: CGAGTAGTCGT-
##
## --- Alignment 2 ---
## Seq1: -GT-TCCAGGTA
## Seq2: CGAGTAGTCGT-
##
## --- Alignment 3 ---
## Seq1: GTTCCAG--GTA
## Seq2: CGAGTAGTCGT-
##
## --- Alignment 4 ---
## Seq1: -GTTCCAG--GTA
## Seq2: CG-AGTAGTCGT-
##
## --- Alignment 5 ---
## Seq1: -GTTCCAG--GTA
## Seq2: CGA-GTAGTCGT-
##
## --- Alignment 6 ---
## Seq1: -GTTCCAG--GTA
## Seq2: CGAG-TAGTCGT-
##
## --- Alignment 7 ---
```

```
## Seq1: -GTTCCAG--GTA
## Seq2: CGAGT-AGTCGT-
##
## --- Alignment 8 ---
## Seq1: -G-TTCCAG--GTA
## Seq2: CGAGT--AGTCGT-
##
## --- Alignment 9 ---
## Seq1: -GT-TCCAG--GTA
## Seq2: CGAGT--AGTCGT-
```

b. **Repeat (a), but this time the penalty for an indel is -0.1.**

```
## Max alignment score: 5.1
## Found 2 optimal alignment(s):
##
## --- Alignment 1 ---
## Seq1: -G--T--TCCAGGTA
## Seq2: CGAGTAGT-C-G-T-
##
## --- Alignment 2 ---
## Seq1: -G--T--TCCAGGTA
## Seq2: CGAGTAGTC--G-T-
```

c. **Repeat (a), using the below substitution matrix and a universal penalty of -0.5 for indels.**

	A	C	G	T
A	2	-2	1	-1
C	-2	1	0	-1
G	1	0	1	-1
T	-1	-1	-1	2

```
## Max alignment score: 4.5
## Found 57 optimal alignment(s):
##
## --- Alignment 1 ---
## Seq1: ---GT--TCCAGGTA
## Seq2: CGAGTAGT-C--GT-
##
## --- Alignment 2 ---
## Seq1: --G-T--TCCAGGTA
## Seq2: CGAGTAGT-C--GT-
##
```

```
## --- Alignment 3 ---
## Seq1: -G--T--TCCAGGTA
## Seq2: CGAGTAGT-C--GT-
##
## --- Alignment 4 ---
## Seq1: ---GT--TCCAGGTA
## Seq2: CGAGTAGTC---GT-
##
## --- Alignment 5 ---
## Seq1: --G-T--TCCAGGTA
## Seq2: CGAGTAGTC---GT-
##
## --- Alignment 6 ---
## Seq1: -G--T--TCCAGGTA
## Seq2: CGAGTAGTC---GT-
##
## --- Alignment 7 ---
## Seq1: --GTTCCAG--GTA
## Seq2: CGAGT--AGTCGT-
##
## --- Alignment 8 ---
## Seq1: -G-TTCCAG--GTA
## Seq2: CGAGT--AGTCGT-
##
## --- Alignment 9 ---
## Seq1: ---GTTCCAG--GTA
## Seq2: CGAG-T--AGTCGT-
##
## --- Alignment 10 ---
## Seq1: --G-TTCCAG--GTA
## Seq2: CGAG-T--AGTCGT-
##
## --- Alignment 11 ---
## Seq1: -G--TTCCAG--GTA
## Seq2: CGAG-T--AGTCGT-
##
## --- Alignment 12 ---
## Seq1: -GT-TCCAG--GTA
## Seq2: CGAGT--AGTCGT-
##
## --- Alignment 13 ---
## Seq1: --GT-TCCAG--GTA
## Seq2: CGA-GT--AGTCGT-
##
## --- Alignment 14 ---
## Seq1: -G-T-TCCAG--GTA
## Seq2: CGA-GT--AGTCGT-
##
```

```
## --- Alignment 15 ---
## Seq1: -GT--TCCAG--GTA
## Seq2: CG-AGT--AGTCGT-
##
## --- Alignment 16 ---
## Seq1: ---GTTCCAG--GTA
## Seq2: CGAGT---AGTCGT-
##
## --- Alignment 17 ---
## Seq1: --G-TTCCAG--GTA
## Seq2: CGAGT---AGTCGT-
##
## --- Alignment 18 ---
## Seq1: -G--TTCCAG--GTA
## Seq2: CGAGT---AGTCGT-
##
## --- Alignment 19 ---
## Seq1: ---GT--TCCAGGTA
## Seq2: CGAGTAGT-C-G-T-
##
## --- Alignment 20 ---
## Seq1: --G-T--TCCAGGTA
## Seq2: CGAGTAGT-C-G-T-
##
## --- Alignment 21 ---
## Seq1: -G--T--TCCAGGTA
## Seq2: CGAGTAGT-C-G-T-
##
## --- Alignment 22 ---
## Seq1: ---GT--TCCAGGTA
## Seq2: CGAGTAGTC--G-T-
##
## --- Alignment 23 ---
## Seq1: --G-T--TCCAGGTA
## Seq2: CGAGTAGTC--G-T-
##
## --- Alignment 24 ---
## Seq1: -G--T--TCCAGGTA
## Seq2: CGAGTAGTC--G-T-
##
## --- Alignment 25 ---
## Seq1: ---GT--TCCAGGTA
## Seq2: CGAGTAGT-CG--T-
##
## --- Alignment 26 ---
## Seq1: --G-T--TCCAGGTA
## Seq2: CGAGTAGT-CG--T-
##
```

```
## --- Alignment 27 ---
## Seq1: -G--T--TCCAGGTA
## Seq2: CGAGTAGT-CG--T-
##
## --- Alignment 28 ---
## Seq1: ---GT--TCCAGGTA
## Seq2: CGAGTAGTC-G--T-
##
## --- Alignment 29 ---
## Seq1: --G-T--TCCAGGTA
## Seq2: CGAGTAGTC-G--T-
##
## --- Alignment 30 ---
## Seq1: -G--T--TCCAGGTA
## Seq2: CGAGTAGTC-G--T-
##
## --- Alignment 31 ---
## Seq1: GTTCCA--GGT-A-
## Seq2: ---CGAGTAGTCGT
##
## --- Alignment 32 ---
## Seq1: --GTTCCAGGT-A-
## Seq2: CGAGT--A-GTCGT
##
## --- Alignment 33 ---
## Seq1: -G-TTCCAGGT-A-
## Seq2: CGAGT--A-GTCGT
##
## --- Alignment 34 ---
## Seq1: ---GTTCCAGGT-A-
## Seq2: CGAG-T--A-GTCGT
##
## --- Alignment 35 ---
## Seq1: --G-TTCCAGGT-A-
## Seq2: CGAG-T--A-GTCGT
##
## --- Alignment 36 ---
## Seq1: -G--TTCCAGGT-A-
## Seq2: CGAG-T--A-GTCGT
##
## --- Alignment 37 ---
## Seq1: -GT-TCCAGGT-A-
## Seq2: CGAGT--A-GTCGT
##
## --- Alignment 38 ---
## Seq1: --GT-TCCAGGT-A-
## Seq2: CGA-GT--A-GTCGT
##
```

```
## --- Alignment 39 ---
## Seq1: -G-T-TCCAGGT-A-
## Seq2: CGA-GT--A-GTCGT
##
## --- Alignment 40 ---
## Seq1: -GT--TCCAGGT-A-
## Seq2: CG-AGT--A-GTCGT
##
## --- Alignment 41 ---
## Seq1: ---GTTCCAGGT-A-
## Seq2: CGAGT---A-GTCGT
##
## --- Alignment 42 ---
## Seq1: --G-TTCCAGGT-A-
## Seq2: CGAGT---A-GTCGT
##
## --- Alignment 43 ---
## Seq1: -G--TTCCAGGT-A-
## Seq2: CGAGT---A-GTCGT
##
## --- Alignment 44 ---
## Seq1: GTTCCAG--GT-A-
## Seq2: ---CGAGTAGTCGT
##
## --- Alignment 45 ---
## Seq1: --GTTCCAGGT-A-
## Seq2: CGAGT--AG-TCGT
##
## --- Alignment 46 ---
## Seq1: -G-TTCCAGGT-A-
## Seq2: CGAGT--AG-TCGT
##
## --- Alignment 47 ---
## Seq1: ---GTTCCAGGT-A-
## Seq2: CGAG-T--AG-TCGT
##
## --- Alignment 48 ---
## Seq1: --G-TTCCAGGT-A-
## Seq2: CGAG-T--AG-TCGT
##
## --- Alignment 49 ---
## Seq1: -G--TTCCAGGT-A-
## Seq2: CGAG-T--AG-TCGT
##
## --- Alignment 50 ---
## Seq1: -GT-TCCAGGT-A-
## Seq2: CGAGT--AG-TCGT
##
```

```
## --- Alignment 51 ---
## Seq1: --GT-TCCAGGT-A-
## Seq2: CGA-GT--AG-TCGT
##
## --- Alignment 52 ---
## Seq1: -G-T-TCCAGGT-A-
## Seq2: CGA-GT--AG-TCGT
##
## --- Alignment 53 ---
## Seq1: -GT--TCCAGGT-A-
## Seq2: CG-AGT--AG-TCGT
##
## --- Alignment 54 ---
## Seq1: ---GTTCCAGGT-A-
## Seq2: CGAGT---AG-TCGT
##
## --- Alignment 55 ---
## Seq1: --G-TTCCAGGT-A-
## Seq2: CGAGT---AG-TCGT
##
## --- Alignment 56 ---
## Seq1: -G--TTCCAGGT-A-
## Seq2: CGAGT---AG-TCGT
##
## --- Alignment 57 ---
## Seq1: GTTCCAG-G-T-A-
## Seq2: ---CGAGTAGTCGT
```

5) **Use the following blocks to construct a substitution matrix using the methodology outlined in the notes. Note that this is using the standard DNA nucleotides (ACGT):**

- **GCATTA**
- **GGGTTT**
- **GCCTTA**
- **AGCTAA**
- **GGGAAC**
- **GCCTAG**

After computing the score matrix using the steps in the slides, we end up with this dataframe:

Table 3: Substitution Matrix Calculations

aligned_pair	frequency	relative_frequency	expected_prob	log_odds	score
A -> A	6	0.0667	0.0625	0.0931	0
A -> C	6	0.0667	0.0972	-0.5443	-1
A -> G	10	0.1111	0.1528	-0.4594	-1
A -> T	17	0.1889	0.1250	0.5956	1
C -> C	6	0.0667	0.0378	0.8182	2
C -> G	16	0.1778	0.1188	0.5812	1
C -> T	1	0.0111	0.0972	-3.1293	-6
G -> G	14	0.1556	0.0934	0.7365	1
G -> T	1	0.0111	0.1528	-3.7814	-8
T -> T	13	0.1444	0.0625	1.2086	2

which yields the following score matrix:

$$S = \begin{array}{c|cccc} & \text{A} & \text{C} & \text{G} & \text{T} \\ \hline \text{A} & 0 & -1 & -1 & 1 \\ \text{C} & -1 & 2 & 1 & -6 \\ \text{G} & -1 & 1 & 1 & -8 \\ \text{T} & 1 & -6 & -8 & 2 \end{array}$$

Appendix

```
knitr::opts_chunk$set(
  dev = "cairo_pdf",
  fig.width = 5,
  fig.height = 5,
  fig.align = 'center',
  echo = FALSE,
  message = FALSE,
  warning = FALSE,
  error = FALSE,
  results = 'markup'
)

# Load required libraries
library("tidyverse")
library("patchwork")
library("glue")
library("scales", warn.conflicts = FALSE)
library("extrafont")
library("tinytex")
library("knitr")
library("tidyr")
library("latex2exp")
library("purrr")
library("MASS")
library("kableExtra")

theme_set(theme_minimal(base_family = "Roboto Condensed"))

conflicted::conflicts_prefer(
  readr::col_factor(),
  purrr::discard(),
  dplyr::lag(),
  readr::parse_date(),
  kableExtra::group_rows(),
  dplyr::select
)

# ------------------- Problem 1a -------------------
gen_transition_matrix <- function(k) {
  C <- diag(0.25, k)
  B <- matrix(0.75, nrow = k, ncol = 1)

  cbind(B, C) |>
    rbind(c(rep(0, k), 1))
}
```

```r
gen_transition_matrix(3)

# ------------------- Problem 1b -------------------
compute_prob <- function(k, n = 250) {
  P <- gen_transition_matrix(k)
  init <- matrix(c(1, rep(0, times = k)), nrow = 1) # 1 x (k+1) row vector

  x <- init
  for (i in seq_len(n)) {
    x <- x %*% P
  }

  x[k + 1]
}

# 1. Compute P(Y >= k) for k = 0 to 11
# Using map_dbl to return a numeric vector instead of a list
probs_geq <- map_dbl(0:11, compute_prob)

# 2. Calculate exact probabilities P(Y = k) = P(Y >= k) - P(Y >= k + 1)
probs_eq <- probs_geq[1:11] - probs_geq[2:12]

# 3. Create a tidy tibble for the output table
prob_table <- tibble(
  k = c(as.character(0:10), "$\\ge 11$"),
  `$P(Y_{(n)} = k)$` = c(probs_eq, probs_geq[12])
)

# 4. Generate the kable table
prob_table |>
  kable(
    format = "latex",
    digits = 6,
    align = "c",
    escape = FALSE, # This prevents kable from escaping your $ and \
    caption = "Partial Mass Function for the Longest Matching Subsequence ($Y_

  )

# ------------------- Problem 2b -------------------

rtrunc_geom <- function(n_sims, p_mismatch = 0.75, max_val = 250) {
  # Draw from truncated geometric using inverse CDF / rejection
  # P(X = k) = (1-p)k * p / (1 - (1-p)(n+1))
  # Equivalently: draw standard geom, reject if > max_val
  q <- 1 - p_mismatch # probability of a match = 0.25

  # Normalizing constant
```

```r
  norm_const <- 1 - q(max_val + 1)

  # Inverse CDF method: F(k) = (1 - q(k+1)) / (1 - q(n+1))
  # Set U = F(k), solve for k: k = floor(log(1 - U*norm_const) / log(q))
  u <- runif(n_sims)
  k <- floor(log(1 - u * norm_const) / log(q))
  pmin(k, max_val) # safety clamp
}


simulate_Y <- function(n = 250, n_reps = 100000, p_mismatch = 0.75) {
  replicate(n_reps, {
    pos <- 0
    max_run <- 0
    while (pos < n) {
      remaining <- n - pos
      # Draw run length from truncated geom with max = remaining - 1
      # (we need at least 1 position for the mismatch, unless we're at the en
      run <- rtrunc_geom(1, p_mismatch, max_val = remaining)
      max_run <- max(max_run, run)
      pos <- pos + run + 1 # +1 for the mismatching position
    }
    max_run
  })
}

set.seed(613)
sim_results <- simulate_Y(n = 250, n_reps = 100000)

# Estimate probabilities P(Y = k) for k = 0:10 and P(Y >= 11)
sim_probs_eq <- sapply(0:10, function(k) mean(sim_results == k))
sim_prob_geq11 <- mean(sim_results >= 11)

# ------------------- Problem 2c ------------------
simulate_Y_mismatches <- function(
  n = 250,
  n_reps = 100000,
  p_mismatch = 0.75,
  max_mismatches = 2
) {
  replicate(n_reps, {
    pos <- 0
    max_run <- 0

    while (pos < n) {
      mismatches <- 0
      match_count <- 0
```

```r
      while (pos < n && mismatches <= max_mismatches) {
        remaining <- n - pos
        # Draw matches until next mismatch (or end of string)
        run <- rtrunc_geom(1, p_mismatch, max_val = remaining)
        match_count <- match_count + run
        pos <- pos + run + 1 # +1 for the mismatch position consumed
        mismatches <- mismatches + 1
      }

      # Record the number of MATCHING positions (excluding mismatches)
      max_run <- max(max_run, match_count)

      # If we overshot due to the +1, we are done
    }
    max_run
  })
}

set.seed(613)
sim_results_2mm <- simulate_Y_mismatches(n = 250, n_reps = 100000)

sim_probs_eq_2mm <- sapply(0:10, function(k) mean(sim_results_2mm == k))
sim_prob_geq11_2mm <- mean(sim_results_2mm >= 11)

sim_table_2mm <- tibble(
  k = c(as.character(0:10), "$\\ge 11$"),
  `0 Mismatches` = c(sim_probs_eq, sim_prob_geq11),
  `2 Mismatches` = c(sim_probs_eq_2mm, sim_prob_geq11_2mm)
)

sim_table_2mm |>
  kable(
    format = "latex",
    digits = 6,
    align = "c",
    escape = FALSE,
    caption = "Estimated PMF of $Y_{(n)}$: 0 vs.~2 Allowed Mismatches"
  )
# ------------------- Problem 3a -------------------
mnc <- function(money, denominations = c(1, 4, 5, 10, 20, 24, 30, 40, 120)) {
  # dp[i+1] stores min coins for amount i (R is 1-indexed, so amount i -> inde
  # Initialize with Inf (impossible), except dp[1] = 0 (amount 0 needs 0 coins
  dp <- rep(Inf, money + 1)
  dp[1] <- 0 # dp[amount + 1], so amount 0 -> index 1

  # Build up from small values to money (bottom-up)
  for (amount in 1:money) {
    for (coin in denominations) {
```

```r
      if (coin <= amount && dp[amount - coin + 1] != Inf) {
        dp[amount + 1] <- min(dp[amount + 1], dp[amount - coin + 1] + 1)
      }
    }
  }

  result <- dp[money + 1]
  return(ifelse(result == Inf, -1, result))
}

# ------------------- Problem 3b -------------------
mnc(72)
mnc(168)

# ------------------- Problem 4a -------------------
x <- "GTTCCAGGTA"
y <- "CGAGTAGTCGT"


seq_align <- function(seq1, seq2, d, mismatch, match) {
  s1 <- strsplit(seq1, "")[[1]]
  s2 <- strsplit(seq2, "")[[1]]
  n <- length(s1)
  m <- length(s2)

  # 1. INITIALIZATION
  F_mat <- matrix(0, nrow = n + 1, ncol = m + 1)
  F_mat[, 1] <- seq(0, n * d, by = d)
  F_mat[1, ] <- seq(0, m * d, by = d)

  # 2. MATRIX FILLING
  for (i in 2:(n + 1)) {
    for (j in 2:(m + 1)) {
      match_score <- ifelse(s1[i - 1] == s2[j - 1], match, mismatch)
      diag_score <- F_mat[i - 1, j - 1] + match_score
      up_score <- F_mat[i - 1, j] + d
      left_score <- F_mat[i, j - 1] + d

      F_mat[i, j] <- max(diag_score, up_score, left_score)
    }
  }

  # 3. TRACEBACK (Recursive)
  all_alignments <- list()

  # Define the recursive search closure
  traceback <- function(i, j, align1, align2) {
    # Base case: Reached the origin (top-left)
```

```
    if (i == 1 && j == 1) {
      # Use <<- to append to the list in the parent environment
      all_alignments <<- append(
        all_alignments,
        list(c(
          paste(align1, collapse = ""),
          paste(align2, collapse = "")
        ))
      )
      return()
    }

    # Branch 1: Diagonal (Match/Mismatch)
    if (i > 1 && j > 1) {
      match_score <- ifelse(s1[i - 1] == s2[j - 1], match, mismatch)
      if (F_mat[i, j] == F_mat[i - 1, j - 1] + match_score) {
        traceback(i - 1, j - 1, c(s1[i - 1], align1), c(s2[j - 1], align2))
      }
    }

    # Branch 2: Up (Deletion in seq2)
    if (i > 1) {
      if (F_mat[i, j] == F_mat[i - 1, j] + d) {
        traceback(i - 1, j, c(s1[i - 1], align1), c("-", align2))
      }
    }

    # Branch 3: Left (Insertion in seq1)
    if (j > 1) {
      if (F_mat[i, j] == F_mat[i, j - 1] + d) {
        traceback(i, j - 1, c("-", align1), c(s2[j - 1], align2))
      }
    }
  }

  # Initialize recursion from the bottom-right cell
  traceback(n + 1, m + 1, character(), character())

  list(
    matrix = F_mat,
    alignments = all_alignments,
    seq1_chars = c("-", s1),
    seq2_chars = c("-", s2)
  )
}


print_alignments <- function(alignment_obj) {
```

```r
  # Extract the max score from the bottom-right cell of the matrix
  mat <- alignment_obj$matrix
  max_score <- mat[nrow(mat), ncol(mat)]

  cat(sprintf(
    "Max alignment score: %g\nFound %d optimal alignment(s):\n\n",
    max_score,
    length(alignment_obj$alignments)
  ))

  alignment_obj$alignments |>
    purrr::iwalk(\(aln, idx) {
      cat(sprintf("--- Alignment %d ---\n", idx))
      cat("Seq1:", aln[1], "\n")
      cat("Seq2:", aln[2], "\n\n")
    })
}

seq_align(x, y, d = -1, mismatch = -0.5, match = 1) |>
  print_alignments()

# ------------------- Problem 4b -------------------
seq_align(x, y, d = -0.1, mismatch = -0.5, match = 1) |>
  print_alignments()
# ------------------- Problem 4c -------------------

sub_matrix <- matrix(
  c(2, -2, 1, -1, -2, 1, 0, -1, 1, 0, 1, -1, -1, -1, -1, 2),
  nrow = 4,
  ncol = 4,
  byrow = TRUE,
  dimnames = list(c("A", "C", "G", "T"), c("A", "C", "G", "T"))
)

seq_align_mat <- function(seq1, seq2, d, sub_mat) {
  s1 <- strsplit(seq1, "")[[1]]
  s2 <- strsplit(seq2, "")[[1]]
  n <- length(s1)
  m <- length(s2)

  # 1. INITIALIZATION
  F_mat <- matrix(0, nrow = n + 1, ncol = m + 1)
  F_mat[, 1] <- seq(0, n * d, by = d)
  F_mat[1, ] <- seq(0, m * d, by = d)

  # 2. MATRIX FILLING
  for (i in 2:(n + 1)) {
    for (j in 2:(m + 1)) {
```

```
    match_score <- sub_mat[s1[i - 1], s2[j - 1]]
    diag_score <- F_mat[i - 1, j - 1] + match_score
    up_score <- F_mat[i - 1, j] + d
    left_score <- F_mat[i, j - 1] + d

    F_mat[i, j] <- max(diag_score, up_score, left_score)
  }
}

# 3. TRACEBACK (Recursive)
all_alignments <- list()

traceback <- function(i, j, align1, align2) {
  # Base case: Reached the origin (top-left)
  if (i == 1 && j == 1) {
    all_alignments <<- append(
      all_alignments,
      list(c(
        paste(align1, collapse = ""),
        paste(align2, collapse = "")
      ))
    )
    return()
  }

  # Branch 1: Diagonal (Match/Mismatch from Substitution Matrix)
  if (i > 1 && j > 1) {
    match_score <- sub_mat[s1[i - 1], s2[j - 1]]
    # Use a tolerance for floating-point equality check
    if (abs(F_mat[i, j] - (F_mat[i - 1, j - 1] + match_score)) < 1e-9) {
      traceback(i - 1, j - 1, c(s1[i - 1], align1), c(s2[j - 1], align2))
    }
  }

  # Branch 2: Up (Deletion in seq2)
  if (i > 1) {
    if (abs(F_mat[i, j] - (F_mat[i - 1, j] + d)) < 1e-9) {
      traceback(i - 1, j, c(s1[i - 1], align1), c("-", align2))
    }
  }

  # Branch 3: Left (Insertion in seq1)
  if (j > 1) {
    if (abs(F_mat[i, j] - (F_mat[i, j - 1] + d)) < 1e-9) {
      traceback(i, j - 1, c("-", align1), c(s2[j - 1], align2))
    }
  }
}
```

```r
  # Initialize recursion from the bottom-right cell
  traceback(n + 1, m + 1, character(), character())

  list(
    matrix = F_mat,
    alignments = all_alignments,
    seq1_chars = c("-", s1),
    seq2_chars = c("-", s2)
  )
}

# Run the alignment and print the outputs
seq_align_mat(x, y, d = -0.5, sub_mat = sub_matrix) |>
  print_alignments()

# ------------------- Problem 5 --------------------
# Step 1
s1 <- "GCATTA"
s2 <- "GGGTTT"
s3 <- "GCCTTA"
s4 <- "AGCTAA"
s5 <- "GGGAAC"
s6 <- "GCCTAG"

seq_vec <- c(s1, s2, s3, s4, s5, s6)

# Step 2: Quantify the frequency with which each nucleotide appears across al
#  samples and all positions.
nuc_freqs <- seq_vec |>
  str_split("") |>
  unlist() |>
  tibble(nucleotide = _) |>
  count(nucleotide, name = "frequency") |>
  mutate(p_i = frequency / sum(frequency))

# Step 3: Calculate the frequency with which each aligned pair occurs within
#  same position.
calc_column_pairs <- function(col_chars) {
  combn(col_chars, 2, simplify = FALSE) |>
    map_chr(~ paste(sort(.x), collapse = " -> ")) |>
    tibble(aligned_pair = _) |>
    count(aligned_pair, name = "count")
}

seq_df <- seq_vec |>
  str_split("") |>
  do.call(rbind, args = _) |>
```

```r
  as_tibble(.name_repair = "minimal")

final_freqs <- seq_df |>
  map(calc_column_pairs) |>
  list_rbind() |>
  group_by(aligned_pair) |>
  summarize(frequency = sum(count), .groups = "drop") |>
  mutate(relative_frequency = frequency / sum(frequency)) |>
  select(aligned_pair, frequency, relative_frequency)

# Step (4): Calculate the expected frequency of these pairs assuming random
#  behavior (i.e., if I am randomly picking nucleotides with replacement
#  from this block, what is the probability I get the pair AA, AG etc).
expected_freqs <- expand_grid(
  nuc1 = nuc_freqs$nucleotide,
  nuc2 = nuc_freqs$nucleotide
) |>
  filter(nuc1 <= nuc2) |>
  left_join(nuc_freqs, by = c("nuc1" = "nucleotide")) |>
  rename(p1 = p_i) |>
  left_join(nuc_freqs, by = c("nuc2" = "nucleotide")) |>
  rename(p2 = p_i) |>
  mutate(
    expected_prob = if_else(nuc1 == nuc2, p12, 2 * p1 * p2),
    aligned_pair = paste(nuc1, "->", nuc2)
  ) |>
  select(aligned_pair, expected_prob)

# Step 5: Calculate Log-Odds Scores
score_matrix_df <- final_freqs |>
  left_join(expected_freqs, by = "aligned_pair") |>
  mutate(
    log_odds = log2(relative_frequency / expected_prob),
    score = round(2 * log_odds)
  )

# Print cleanly to LaTeX
score_matrix_df |>
  kable(
    format = "latex",
    digits = 4,
    booktabs = TRUE,
    caption = "Substitution Matrix Calculations"
  )
```