HW1 HGEN 48800 Lin_Yu

Lin_Yu

2024-03-31

Problem 1

(a)

i. Implement the linear search algorithm

```
# Implement the linear search algorithm, which scans through a given sequence, looking
linear_search <- function(arr, target) {
  n <- length(arr)
  for (i in 1:n) {
    if (arr[i] == target) {
      return(i)  # Return the index where the target is found
    }
  }
  return("NULL")  # If target is not found, return "NULL"
}
```

```
# simple test:
arr <- c(1, 5, 3)
target <- 5
result <- linear_search(arr, target)
if (result == "NULL") {
  cat("Target", target, "not found in array.")
} else {
  cat("Target", target, "found at index", result)
}
```

```
## Target 5 found at index 2
```

ii.Loop invariant to prove the correctness of our algorithm:

-Base case: At iteration 0, the algorithm checks if A[1] equals v. If it does, the algorithm returns the index 0. Otherwise, it returns "Null".

-Induction Step: Assuming the algorithm works correctly for the first (k-1) iterations without finding v, at the kth iteration, it checks if A[k] equals v. If v is not found among the first k elements, it is not in the array. If v is found at index k, the algorithm returns k.

Therefore, the algorithm operates correctly for all k iterations.

(b) i.The average number of elements needed to be checked:

- If the one we are searching for is equally likely to be any element in sequence A, let X be the index of the element we search for, Y be the number of elements we checked. Then we know Y=X The average number of elements needed to be checked

$$E(Y) = \sum_{i=1}^{n} p(X = i)X = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

  ii.The number of elements needed to be checked in worset case: The worst case: what we are looking for is a_n, we need n iterations.

iii.The average-case running time and the worst case should be similar, as O(n).

(c)

- Sort the sequence A

```
Insertion_sort <- function(arr) {
  n <- length(arr)
  for (j in 2:n) {
    key <- arr[j]
    i <- j-1
    while (i>0 && arr[i] >key ) {
      arr[i+1] <- arr[i]
      i=i-1
    }
      arr[i+1] <- key
  }
  return(arr)
  }
```

-implement binary search

```
binary_search <- function(arr,target){
  n <- length(arr)
  a=1
  b=n
```

```r
  while (a<=b){
    mid <-  floor((b+a)/2)
    if (target == arr[mid]){
      return(mid)
    }else if (arr[mid] < target) {
      a=mid+1
    }else{
      b=mid-1
    }
  }
  return("NULL")  # If target is not found, return "NULL"

}
```

```r
# test:
arr <- c(1, 3, 13, 5, 15, 17, 19)
arr <- Insertion_sort(arr)
print(arr)
```

```
## [1]  1  3  5 13 15 17 19
```

```r
target <- 13
result <- binary_search(arr, target)
if (result == "NULL") {
  cat("Target", target, "not found in array.")
} else {
  cat("Target", target, "found at index", result, "in the sorted sequence of A")
}
```

```
## Target 13 found at index 4 in the sorted sequence of A
```

(d)

The worst-case should be when target element is not in the array, but lies in the range of $[a_1, a_n]$. The worst-case time complexity of binary search is O(log n), where n is the number of elements in the array

Reasoning:

- As the algorithm will continue to divide the array in half until it reaches a point where the subarray to be searched is empty, let k be the number of iterations needed,

$$n/2^k = 1$$

$$k = \log_2 n$$

Problem 2

It is in the handwritten section :)

Problem 3

Merge-sort Algorithm

i. implement the merge-sort algorithm for the sorting problem:

```r
merge <- function(arr, p, q, r) {
  n1 <- q - p + 1
  n2 <- r - q
  L <- arr[p:q]
  R <- arr[(q + 1):r]
  inf <- max(arr) * 100

  L <- c(L, inf)
  R <- c(R, inf)

  i <- 1
  j <- 1
  for (k in p:r) {
    if (L[i] <= R[j]) {
      arr[k] <- L[i]
      i <- i + 1
    } else {
      arr[k] <- R[j]
      j <- j + 1
    }
  }
  return(arr)
}
```

```r
merge_sort <- function(arr, p, r) {
  if (p < r) {
    q <- floor((p + r) / 2)
    arr=merge_sort(arr, p, q)
    arr=merge_sort(arr, q + 1, r)
    merge(arr, p, q, r)
  }else{
    arr
  }
}

# Test the merge-sort function
arr1 <- c( 4,3,1,3,8,7,2)
sort_arr=merge_sort(arr1,1,length(arr1))
print(sort_arr)
```

```
## [1] 1 2 3 3 4 7 8
```

Problem 4

i.pseudo code:

```
Function hash_sequence(tuple):
//Assign 3-tuple a value (hash function)
    A <- 0
    C <- 1
    G <- 2
    T <- 3
    hash_value <- 0
    hash_value += value_of(tuple[0]) * (4 ** 2)
    hash_value += value_of(tuple[1]) * (4 ** 1)
    hash_value += value_of(tuple[2]) * (4 ** 0)
    return hash_value

Function positions_table(sequence1, sequence2, ..., sequencek):
// Generate the position_table using the provided sequences as our dataset

    positions_table <- empty position tables
    for j from 1 to k:
    // Iteration over all sequences j, j=1,2,..k
        for i from 1 to length(sequence j) - 2:
        // check all the 3-tuple and store them into the position table
            tuple <- substring of sequencej from index i to i + 2
            hash_value <- hash_sequence(tuple)
            // Compute hash value for this tuple
            if (hash_value is in the positions_table):
               Append index (j, i) to positions_table[hash_value]
            else:
               create a new row in the positions_table
    return positions_table

Function search_hash_table(target_sequence, positions_table):
```

```
// This function generates a hash_table based on the target sequence and
// determines if the target sequence is found.

    m <- length(target_sequence) / 3
    hash_table <- empty data frame
    for s from 0 to m - 1:
        tuple <- substring of target_sequence from index 3 * s + 1 to 3 * s + 3
        hash_value <- hash_sequence(tuple)
        for index(j, i) stored in positions_table[hash_value]:
            append tuple (j, i - 3 * s, i) as Sequence_index,target_start_Position
            and start_Position to hash_table
    M <- sorted hash_table
    for each m consecutive tuples(rows) in M:
        if all tuples(rows) have the same first index:
           if all rows have the same second index:
              return the first tuple of the m consecutive tuples(rows)
    return "Pattern not found"
```

ii. Implementing the pseudo code

```r
# Assign 3-tuple a value (hash function)
hash_sequence <- function(tuple) {
  nucleotides <- c("A" = 0, "C" = 1, "G" = 2, "T" = 3)
  hash_value <- 0
  for (i in 1:3) {
    hash_value <- hash_value + nucleotides[substring(tuple, i, i)] * (4 ** (3 - i))

  }
  return(as.integer(hash_value))
}

#Test if the hash_sequence works
hash_sequence("CGA")
```

```
## [1] 24
```

```r
# Create an empty data frame with appropriate column names
positions_table <- data.frame(Hash_Value = numeric(),
                              Substring = character(),
                              Position_Index = I(list()),
                              stringsAsFactors = FALSE)

# Generate the position_table using the provided sequences as our dataset
```

```
build_positions_table <- function(...) {
  sequences <- list(...)

  for (j in 1:length(sequences)) {
    for (i in 1:(nchar(sequences[[j]]) - 2)) {
      substring <- substr(sequences[[j]], i, i + 2)  # Extract substring from the seque
      hash_value <- hash_sequence(substring)

      # Check if the hash value already exists in the positions table
      if (hash_value %in% positions_table$Hash_Value) {
        # If yes, update the corresponding row
        idx <- which(positions_table$Hash_Value == hash_value)
        positions_table$Position_Index[[idx]] <- c(positions_table$Position_Index[[idx]]
      } else {
        # If no, create a new row
        new_row <- data.frame(Hash_Value = hash_value,
                              Substring = substring,
                              Position_Index = I(list(paste("(", j, ",", i, ")", sep = "
        positions_table <- rbind(positions_table, new_row)
      }
    }
  }

  return(positions_table)
}
# Example usage
sequence1 <- "GCTGCT"
positions_table <- build_positions_table(sequence1)
```

```
#convert string "(j,i)" to numerical list (j,i)
translate_string <- function(string) {
  # Remove parentheses and split the string by comma
  components <- strsplit(gsub("[()]", "", string), ",")[[1]]

  # Convert components to numeric and store as a list
  numeric_pair <- list(j = as.numeric(components[1]), i = as.numeric(components[2]))

  return(numeric_pair)
}

x=translate_string("(3,6)")

#Create the hash_table based on target_sequence and positions_table
hash_table <- function(target_sequence, positions_table) {
```

```r
  m <- nchar(target_sequence) / 3
  #create an empty data frame for hash_table
  M <- data.frame(Sequence_index = numeric(),
                  target_start_Position = numeric(),
                  start_Position = numeric(),
                  stringsAsFactors = FALSE)

  # Loop through substrings of target_sequence
  for (s in 0:(m - 1)) {
    start <- 3 * s + 1
    end <- 3 * s + 3
    target_tuple <- substr(target_sequence, start, end)# extract(s+1)th 3-tuples in the
    hash_value <- hash_sequence(target_tuple)

    # Check if hash_value of target_tuple exists in positions_table
    if (hash_value %in% positions_table$Hash_Value) {
      idx <- which(positions_table$Hash_Value == hash_value)
      position_indexes=positions_table$Position_Index[[idx]]
      for (indexes in position_indexes) {
        index <- translate_string(indexes)
        # Append (j, i - 3 * s, i) to hash_table
        new_row <- data.frame(Sequence_index = as.numeric(index[1]),
                              target_start_Position = as.numeric(index[2]) - 3 * s,
                              start_Position = as.numeric(index[2]))
                              # the place where the first letter in target tuple occurs
        M <- rbind(M, new_row) # Add new row to M
      }
    }
  }
  return(M)
}

M=hash_table('GCT', positions_table)
```

```r
# This function generates the hash_table based on target sequence and
# determines if it founds the target sequence
search_hash_table <- function(target_sequence,positions_table) {
  m <- nchar(target_sequence) / 3
  M <- hash_table(target_sequence, positions_table)
  sorted_M <- M[order(M$Sequence_index, M$target_start_Position, M$start_Position), ]
  found <- 0
  # Check for m consecutive tuples in M
  for (i in 1:(nrow(sorted_M) - m + 1)) {
    #Check if the first and second indexes of the tuple are the same
    if (all(sorted_M$Sequence_index[i:(i + m - 1)] == sorted_M$Sequence_index[i]) &&
        all(sorted_M$target_start_Position[i:(i + m - 1)] == sorted_M$target_start_Posit
      cat("Our target occurs at sequence", sorted_M$Sequence_index[i], "starting from po
      found <-found+1
      #return(invisible(NULL))  # Exit the function if pattern is found
    }
  }
  if (found==0){
    cat("Pattern not found\n")
  }
  # Print "Pattern not found" if no match is found
}
```

```
target="TAGCTAGCT"
S1 = "GCTGCTGCTGCTAAACGTTTGGGGCAGTCGAT"
S2 = "GGTGCTCCAAGCTTTTGAGTCTGCTAGTGTCAACCCT"
S3 = "GTGGGCCCCCTAGCTAGCTAGCTGGGGCAC"
S4 = "TGTCGCTGGCTGGACTGCTGATCGTAGTAG"
positions_table <- build_positions_table(S1,S2,S3,S4)
search_hash_table(target,positions_table)
```

```
## Our target occurs at sequence 3 starting from position 11
## Our target occurs at sequence 3 starting from position 15
```

HW 1

Problem 2:

(a). $T(n) = a \cdot T(\frac{n}{b}) + f(n)$

i. $T(n) = 2 \cdot T(\frac{n}{4}) + 1$

$a = 2$, $b = 4$, $f(n) = 1$

$n^{\log_b a} = n^{\log_4 2} = n^{\frac{1}{2}}$

$f(n) = 1 = O(n^{\frac{1}{2} - \frac{1}{2}})$. by case (1) of master theorem

$T(n) = \Theta(n^{\log_4 2}) = \Theta(n^{\frac{1}{2}})$

ii. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

$a = 2$ $b = 4$ $f(n) = n^{\frac{1}{2}}$

$n^{\log_b a} = n^{\frac{1}{2}}$, $f(n) = O(n^{\log_a b})$, we apply case 2

$T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n^{\frac{1}{2}} \cdot \log n)$

iii. $T(n) = 2T(\frac{n}{4}) + n$

$a = 2$ $b = 4$ $f(n) = n$

$f(n) = \Omega(n^{\log_b a}) = \Omega(n^{\frac{1}{2}})$. $2 \cdot f(\frac{n}{4}) \le \frac{1}{2} \cdot f(n)$, $\because 2 \cdot \frac{n}{4} \le \frac{1}{2} \cdot n$

case (3) applies , $T(n) = \Theta(f(n)) = \Theta(n)$

iv. $T(n) = 2T(\frac{n}{4}) + n^2$

$a = 2$, $b = 4$, $f(n) = n^2$ $n^{\log_b a} = n^{\frac{1}{2}}$

$f(n) = \Omega(n^{\frac{1}{2}})$ and $2 f(\frac{n}{4}) \le \frac{1}{2} f(n)$ $\because 2 \cdot (\frac{n}{4})^2 \le \frac{1}{2} \cdot n^2$

case (3) applies, $T(n) = \Theta(f(n)) = \Theta(n^2)$

(b) Binary Search for question (1)

$T(n) = T(\frac{n}{2}) + 1$

$a = 1$ $b = 2$ $f(n) = 1$

$n^{\log_2 1} = n^0 = 1$ $\Rightarrow$ $f(n) = O(n^{\log_2 1})$

case (2) applies, $T(n) = \Theta(n^{\log_2 1} \cdot \log n) = \Theta(\log n)$