Correctness

Monday, 30 May, 2022 16:08

Correctness

State a useful invariant for *odd_prod*. Use that invariant to prove that *odd_prod* calculates the product of all odd numbers in *L*.

5 Marks

Note that the empty product (i.e. multiplying no numbers together) is 1.

```
def odd_prod(L[1...n]):
  prod = 1
  i = 1
  while i < len(L)
    if L[i] % 2 == 1:
      prod *= L[i]
    i += 1
  return prod
```

In the exam, there's 3 things to take note while answering:

1. Initialization → Proving that the invariant holds at the beginning *(in this example, it'll be before the loop)*
2. Maintenance → Proving that the invariant holds true throughout the algorithm *(in this example, it'll be inside the loop)*
3. Termination → Proving that the invariant implies correctness of algorithm after termination *(in this example, it'll be after the loop)*

invariant (5 marks)

- 1 m — what is the invariant ? list [1... i-1]
- 0.5 m — explain — initialization (before loop) [1... 0] empty
- 3 m — explain — maintenance — cases
- 0.5 m — explain — termination (after loop) [1-- N]

\# initialization

loop :

\# invariant list [1 ... i-1]

do something

\# invariant list [1 .. i]

\# termination

⇒ if you don't know the invariant, just write a simple invariant and proof it right

⇒ do not use N in invariant, use i

```
def parity (L[1...n])
  p = 0
  i = 1
① # p is the parity of the sum of list [1... i-1]
  while i <= n
     # assume invariant is true [1. .. i-1]
     if L[i] % 2 == 1
        p = not p
     i += 1
     # prove invariant is true here [1... i]
  return p
```

② before the loop, i = 0

therefore list [1...0], is empty parity of an empty list is 0 ; which == p

i = 1

list [1... i-1]

list [1... 0] = empty

③ Assume true for k : invariant is true list [1...k] = list [1... i-1]

prove for k+1 : list [1... k+1] == list [1.. i]

⇒ Prove for each case (if - else)

list [i] is parity positive

list [i] is parity negative

conclusion invariant holds for ...

④ loop terminate : i = n+1

invariant is true for list [1... i-1]

= list [1... n+1-1] = list [1... n]

which is my truth

Studio03 Q4

Problem 4. Write pseudocode for insertion sort, except instead of sorting the elements into non-decreasing order, sort them into non-increasing order. Identify a useful invariant of this algorithm.

```
def insertion(list (1...N)):
  for i from 2 to N:          <- loop invariant begin list [1.....i-1] is sorted
    key = list[i]
    for j from i-1 to 1:
      if list[j] < key:
        swap list[j] , list[j+1]
  list[j+1] = key             <- loop invariant end list [1. ... i] is sorted
```

same

Start: i =2

Maintenance: i = k; assume invariant begin is

End: i = N

Start

i = 2

list [1 .....1] is sorted

assume invariant hold at i = k

① writing algo (pseudo code)

② start / base case

③ maintenance

list[1 .....1] is sorted

assume invariant hold at i=k

list[1....k-1] is sorted

ensure that

inner loop

shift item[j] to the right > list[i]

list[1....k] is sorted

put key in place in list[j]

end i = N

start list[1....N-1] is sorted

⋮

end list[1....N] is sorted

(2) start / base case

(3) maintenance

- invariant at the start and the end.

what it do is in between

- how it ensures that invariant is maintained.

(4) end.

- invariant at the end.

Selection/Insertion Invariant

Tuesday, 31 May, 2022 01:49

Sorting
Selection Sort

MONASH University

- Correctness
  - Loop invariant
    - my_list[0…i-1] is sorted
    - my_list[0…i-1] <= my_list[i…N]
  - Termination
    - i and j always increment and both reach the end of the list
  - So why is it working then?
    - i keep increment till n and we know from invariant 0…i-1 is sorted, thus we will sort the entire list!

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

21

Sorting
Insertion Sort

MONASH University

- Correctness
  - Loop invariant
    - my_list[0…i-1] sorted
  - Termination
    - Simple, I skip this
- Complexity

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

Algorithm Analysis and Sorting

Monday, 30 May, 2022 16:08

	Best	Worst	Average	Stable?	In-place?
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No	Yes
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	No	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	Yes	No
Quick Sort	$O(N \log N)$	$O(N^2)$ – can be made $O(N \log N)$	$O(N \log N)$	Depends	No

Sorting Algo Space Complexity

Monday, 30 May, 2022 23:54

Selection Sort **Space Complexity: O(n)** **Aux Space Complexity: O(1)** • Swapping elements does not create memory • Even if a temporary variable is created, value assigned is not an array whatsoever. O(1) space • Input given is a list of size n. O(n) space.	**Merge Sort** **Space Complexity: O(n)** **Aux Space Complexity: O(n)** • List given as input, O(n) space • Recursive calls on same function, O(n) space
Radix Sort **Space Complexity: O(KN)** **Aux Space Complexity: O(N)** • Each counting sort requires O(N+M) • Aux O(N) because we can just copy the memory address of each value in the count array.	**Recursive Algorithms?** **Aux space == recursive depth** • Every function call, memory is created (recalling MIPS from FIT1008) • Every function call puts three things onto the stack: $fa pointer, parameters, return address. • This does not happen to normal functions as these three things are put on the stack once.

Counting and Radix sort is stable.

Counting Sort
Complexity

- Time?
  - Find the maximum O(N)
  - Build the count-array O(M) where M is the max
  - Go through input list and update the count-array
    - How to make it fast?
    - Therefore this is O(N) since we can have O(1) access to the count-array
  - Loop through count-array to rebuild the original list O(M)
  - Total = O(N + M + N + M + N) = O(N+M)
  - So we want M << N for this to be good
  - If we are doing alphabets only, then the M = 26 for the 26 character (after ascii conversion + maths)

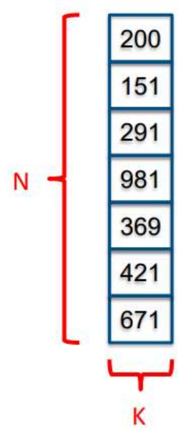
47

- Space?
  - Input list O(N)
  - Count-array O(M)
  - Total = O(N + M)
  - Auxiliary = O(M)

If we want it to be stable the space complexity would not be added it would still be O(N+M).

Radix Sort
Complexity

- What is the complexity?
  - Better than merge sort O(k N log N)!
  - But we know M = 10 for 0, 1, ..., 9
  - Time
    - O(KN + KM) ≈ O(KN)
      where M is the number of unique characters
    - Why? Recall counting sort, we account for the max giving us O(N+M)
    - Then we have K columns giving us O(K) * O(N+M)
  - Space
    - Input is O(KN)
    - Each counting sort needs O(M+N)
    - Total is O(KN + M + N) ≈ O(KN)
    - Auxiliary is O(M + N) ≈ O(N)



140

K since it is columns can be also equal to $\log_b M$
# of columns.

We only need M space and N space because M is the base and since it gets the references for N, it doesn't need to append the whole string or number as reference.