3500 hw 7

Casey Provitera

March 2024

1 Question 1 (30 points)

Consider a sorted array A of n distinct integers. Provide an algorithm to identify an index i where 1 <= i <= n and A[i] = i , given that such an index exists. If multiple such indices exist, the algorithm can return any one of them. Use a divide and conquer approach to solve the problem. Ensure that your algorithm achieves a time complexity better thanO(n) where n represents the number of integers in the array.

1. Write a clear divide and conquer algorithm to solve the problem.

```
def search(A, low, high):
    mid = (low+high)//2
    if (A[mid] == mid):
        return mid
    if (A[mid] < mid):
        search(A, mid+1, high)
    if (A[mid] > mid):
        search(A, low, mid-1)
```

2. Provide the recurrence relation that describes the time complexity of your algorithm.
   The recurrence relation that describes this algorithm is T(n) = 1*T(n/2)+1
   This is because every time we recurs we only create 1 sub problem with a size of half the original sub problem. And we only have +1 for the time to divide and combine, since we are never combining and dividing has a constant run time.

3. Analyze the time complexity of your algorithm. If applicable, use the Master Theorem.
   We Can use the master theorem to solve this time complexity.
   a = 1, b = 2, f(n) = 1
   $log_b a = log_2 1 = 0$
   n^0 = f(n), case 2, all calls equal size
   meaning final time complexity = O(f(n)*log n) = O(log n)

2 Question 2 (70 points)

The objective of this assignment is to implement Quicksort with two different pivot selection strategies and analyze their performance on various input sizes and types. Tasks:

1. Implement Quicksort in your preferred programming language (e.g., Python) with the following pivot selection strategies:
   (a) Pick the last element as the pivot.
   (b) Pick a random element from the list as the pivot.
2. Test both implementations of Quicksort on different types of inputs (Sorted list, Reverse sorted list, Random list) and on different sizes (100, 500, 1000, 1500, and 2000) and measure the running time of each Quicksort implementation for each input size and type.

Submission:

1. (50 points) Submit a report that includes:
   (a) A table showing the running time for each pivot selection strategy, input size, and input type.

List qualities		Quicksort method	Time to sort
Type	Size	of solution	
Sorted	100	Last element pivot	0
Sorted	100	Random element pivot	0
Reverse Sorted	100	Last element pivot	0
Reverse Sorted	100	Random element pivot	0
Random	100	Last element pivot	0
Random	100	Random element pivot	0
Sorted	500	Last element pivot	0.001980543
Sorted	500	Random element pivot	0.001016617
Reverse Sorted	500	Last element pivot	0.002500772
Reverse Sorted	500	Random element pivot	0.001027346
Random	500	Last element pivot	0.00100255
Random	500	Random element pivot	0
Sorted	1000	Last element pivot	0.006597757
Sorted	1000	Random element pivot	0.001006126
Reverse Sorted	1000	Last element pivot	0.009518147
Reverse Sorted	1000	Random element pivot	0.002997398
Random	1000	Last element pivot	0.001504898
Random	1000	Random element pivot	0.00052166
Sorted	1500	Last element pivot	0.014636993
Sorted	1500	Random element pivot	0.003017902
Reverse Sorted	1500	Last element pivot	0.020633698
Reverse Sorted	1500	Random element pivot	0.003514528
Random	1500	Last element pivot	0.000984907
Random	1500	Random element pivot	0.001017094
Sorted	2000	Last element pivot	0.028199673
Sorted	2000	Random element pivot	0.003525019
Reverse Sorted	2000	Last element pivot	0.040918827
Reverse Sorted	2000	Random element pivot	0.003983021
Random	2000	Last element pivot	0.001999378
Random	2000	Random element pivot	0.001986027

(b) A conclusion based on the results obtained from the experiments.
The conclusions I have come to from the results in the above table are as follows:
As one would expect as the size of the list gets larger the time for quicksort to sort the list goes up across the board.
When it comes to what method of picking a pivot for quicksort works the best, data clearly shows that picking a random pivot works the best when we are given a sorted or reverse sorted array. But when we are given a random list picking the last element or a random element as a pivot does change the time to execute but not drastically enough for either to be implemented over the other option.
And finally referring to what type of list quicksort sorts the fastest, it is clear that the algorithm does much better with random lists than lists that are either sorted or reversed. This is because when given a sorted or reversed list quick sort potentially partitions into very unequal portions making the recursive tree larger and lengthening the execution time.

2. (20 points) Submit your implementation code.

```
# first quicksort uses the last element as a pivot
def qsortLast(A, low, high):
    if (low < high):
    # most of the work is done in here
        pivot = partition(A, low, high)
        qsortLast(A, low, pivot-1)
        qsortLast(A, pivot+1, high)

# second qsort uses a random element as a pivot
def qusortRandom(A, low, high):
    if (low < high):
    # most of the work is done in here
        pivot = randpartition(A, low, high)
        qusortRandom(A, low, pivot-1)
        qusortRandom(A, pivot+1, high)

# from lecture slides
def partition(L, i, j):
    pivot = j -1
    j = pivot -1
    while i < j :
        #Pivot all items between left and right
        while L[i] < L[pivot]:
            i = i + 1
        while i<j and L[j] >= L[pivot]:
            j = j - 1
```

```
        if i < j:L[i], L[j] = L[j], L[i]
    #Swap pivot and i
    if L[i]>= L[pivot]:
        L[pivot], L[i] = L[i], L[pivot]
        pivot = i
    return pivot

def randpartition(A, low, high):
    # works by just swaping a random element with the
    # last element and then calling normal partition
    i = random.randint(low, high)
    temp = A[high]
    A[high] = A[i]
    A[i] = temp
    return partition(A, low, high)

# main
if __name__ == "__main__":
    import time
    import random
    import sys
    sys.setrecursionlimit(3000)
    ranges = [100, 500, 1000, 1500, 2000]

    print("Sorted Tests time for quick sorts")
    for value in ranges:
        list1 = [i for i in range(value)] # use for first quick sort
        list2 = [j for j in range(value)] # use for second quick sort
        start = time.time()
        qsortLast(list1, 1, len(list1)-1)
        end = time.time()
        print(f"Pivot as the last element for size \
                {value} total time was: {end-start}")
        start = time.time()
        qusortRandom(list2, 1, len(list2)-1)
        end = time.time()
        print(f"Pivot as a random element for size \
                {value} total time was: {end-start}")

    print("\n\nReversed Tests time for quick sorts")
    for value in ranges:
        list1 = [i for i in range(value,0,-1)] # use for first quick sort
        list2 = [j for j in range(value,0,-1)] # use for second quick sort
        start = time.time()
        qsortLast(list1, 1, len(list1)-1)
        end = time.time()
```

```
        print(f"Pivot as the last element for size \
                {value} total time was: {end-start}")
        start = time.time()
        qusortRandom(list2, 1, len(list2)-1)
        end = time.time()
        print(f"Pivot as a random element for size \
                {value} total time was: {end-start}")

    print("\n\nUnsorted tests time for quick sorts")
    for value in ranges:
        list1 = random.sample(range(0, value), value)
        list2 = list1.copy() # list2 now has the same values as list1
                             # but is an independent copy
        start = time.time()
        qsortLast(list1, 1, len(list1)-1)
        end = time.time()
        print(f"Pivot as the last element for size \
                {value} total time was: {end-start}")
        start = time.time()
        qusortRandom(list2, 1, len(list2)-1)
        end = time.time()
        print(f"Pivot as a random element for size \
                {value} total time was: {end-start}")
```

Note: Python uses a maximum recursion depth of 1000 to ensure no stack overflow errors and infinite recursions are possible. You can change the maximum recursion depth in Python. To do this, call the sys.setrecursionlimit() function. For example, let's set the maximum recursion depth to 3000:
import sys
print(sys.setrecursionlimit(3000))