Name: Homework Solutions.

Login:

Signature:

ECE 368 Spring 2016.

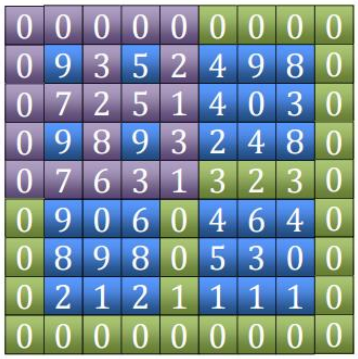
Homework 2

1) Finding Peaks in a $n \times n$ matrix (10pts).

…

However, there is a more efficient peak finding algorithm. The algorithms is as follows:

1. Look at the middle row and column, and the boundaries
2. Find the maximum within these rows/columns
3. If it is a peak (larger than all 4 neighbors):
   - return element
4. Else:
   - Look at larger neighbor
   - Go to step 1 with the quadrant.



1. **Correctness**:
   -If you enter a quadrant, this means that the maximum element on the border of the quadrant is not a peak.
   -When the algorithm stops inside a quadrant, the peak is internal to the quadrant so it is a peak in the array

2. **Complexity**:
   1. Given $nxn\ matrix, you\ will\ look\ into\ middle\ row(n), middle\ column(n) and\ borders(\ c * n)$ $= n + n + cn.$
   2. Then call function recursively in $\frac{n}{2}\ x \frac{n}{2}\ matrix. = middle\ row\left(\frac{n}{2}\right)$, middle column($\frac{n}{2}$) and new borders $(c * \frac{n}{2})$.
   3. Total complexity of function is:
      - $F(n) = F\left(\frac{n}{2}\right) + cn.$
      - $F\left(\frac{n}{2}\right) = F\left(\frac{n}{4}\right) + c\frac{n}{2}$. Similarly $F\left(\frac{n}{4}\right) = F\left(\frac{n}{8}\right) + c\frac{n}{4}$ …
      - $Writting\ in\ terms\ of\ F(n): F(n) = F(1) + c\left(2 + 4 + .. + \frac{n}{4} + \frac{n}{2} + n\right)$
      - $Largest\ term\ is\ n, so\ function\ is\ \boldsymbol{O(n)}$

Greedy Ascend Algorithm (10 pts).

Describe the greedy ascend algorithm and come up with a 5x5 matrix where greedy ascend will stop at the 23erd element.

19	20	21	22	23
18	17	16	15	14
3	2	1	12	13
4	0	0	11	10
5	6	7	8	9

2) **Stacks and Queues (10 pts):**

One common interview question is to show how to create a stack from other abstract data types(ADT). Your task is to write pseudocode to implement a stack using two queues with the following primitives of queues: EMPTY(Q), ENQUEUE (Q, element), DEQUEUE(Q). Your stack should have the following primitives: *PUSH (S, element) and POP(S).* What is the time complexity for each of the operations PUSH (s, element) and POP(s)?

Push(s, e)

ENQUEUE(Q1 ,e)

Pop(s)

if EMPTY(Q1) **(ALWAYS CHECK)**

return error

while not EMPTY(Q1)

temp = pop(Q1);

if EMPTY(Q1) (temp = last element)

ENQUEUE all elements from Q2 to Q1

return temp

else

ENQUEUE(Q2, temp)

Complexity: Push = O(1), POP = O(n)

Solution 2 (POP efficient):

Push(s,e)

ENQUEUE(Q2,e)

ENQUEUE all elements from Q1 to Q2

ENQUEUE all elements from Q2 to Q1

Pop(s)

if EMPTY(Q1) **(ALWAYS CHECK)**

return error

else

return DEQUEUE(Q1)

Complexity: Push = O(n), POP = O(1)

3) Analysis of algorithms (10 pts):

INSERTION_SORT($A[1..n]$)	Cost	Times
1. for $j \leftarrow 2$ to n	C_1	n
2. key $\leftarrow A[j]$	C_2	$n-1$
3. $i \leftarrow j-1$	C_3	$n-1$
4. while $i>0$ and $A[i]>key$	C_4	$\sum_{j=2}^{n} t_j$
5. $A[i+1] \leftarrow A[i]$	C_5	$\sum_{j=2}^{n}(t_j-1)$
6. $i \leftarrow i-1$	C_6	$\sum_{j=2}^{n}(t_j-1)$
7. $A[i+1] \leftarrow$ key	C_7	$n-1$

(a) Let t_j denote the number of times the while loop test in line 4 is executed for that value of j. Fill in for each line of instruction, the number of times the instruction is executed.

(b) Derive the expression for the running time of INSERTION_SORT in terms of n, C_i, and t_j.

Let $T(n)$ = running time of INSERTION_SORT.

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} t_j + C_5 \sum_{j=2}^{n} (t_j - 1) + C_6 \sum_{j=2}^{n} (t_j - 1) + C_7(n-1)$$

(c) What is t_j for the best-case scenario, i.e., when the running time of the algorithm is the smallest. Use that to derive the expression for the best-case running time of INSERTION_SORT in terms of n and C_i. What is the best-case time complexity of INSERTION_SORT using the big-O notation?

The array is already sorted. All t_j are 1. Therefore,

$$\begin{aligned} T(n) &= C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_7(n-1) \\ &= (C_1 + C_2 + C_3 + C_4 + C_7)n - (C_2 + C_3 + C_4 + C_7). \end{aligned}$$

$T(n) = O(n)$. (In fact, $T(n) = \Theta(n)$.)

(d) What is t_j for the worst-case scenario, i.e., when the running time of the algorithm is the largest. Use that to derive the expression for the worst-case running time of INSERTION_SORT in terms of n and C_i. What is the worst-case time complexity of INSERTION_SORT using the big-O notation?

The array is in reverse sorted order. We have to compare the j-th element with the previous $(j-1)$ elements. We also need an additional test to get out of the while-loop. Therefore, $t_j = j$. Hence,

$$\begin{aligned} T(n) &= C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} j + C_5 \sum_{j=2}^{n} (j-1) + C_6 \sum_{j=2}^{n} (j-1) + C_7(n-1) \\ &= C_1 n + C_2(n-1) + C_3(n-1) + C_4 \left(\frac{n(n+1)}{2} - 1 \right) + C_5 \frac{n(n-1)}{2} + C_6 \frac{n(n-1)}{2} + C_7(n-1) \\ &= \left(\frac{C_4 + C_5 + C_6}{2} \right) n^2 + \left(C_1 + C_2 + C_3 + \frac{C_4 - C_5 - C_6}{2} + C_7 \right) n - (C_2 + C_3 + C_4 + C_7). \end{aligned}$$

$$T(n) = O(n^2). \text{ (In fact, } T(n) = \Theta(n^2).)$$

4) Recursive Algorithms (10 pts):

The following C++ function *permute()* prints all permutations of the given string. For example, a call of *permute(0,2)* on "ABC" should print the following (order does not matter).

ABC ACB BAC BCA CBA CAB

Complete the following code and briefly explain what happens to the string in every recursive call. (Note: you shouldn't need to write more than 5 lines of code).

```
#include <iostream>
using namespace std;

char str[] = "ABC";

void swap (char *x, char *y){
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

void permute(int i, int n){
  int j;
  if (i==n){
      cout << str << " ";
  }else{
      for(j=i;j<=n;j++)
      {
        //your code goes here
        swap((str + i), (str+j));
        permute(i+1, n);
        swap((str+i),(str+j));
      }


    }
}
```