Homework 3

Tushar Jain, N12753339
CSCI-GA 1170-003 - Fundamental Algorithms

October 25, 2017

Problem 1.a. Design a recursive algorithm `TROMINO(`n`)` that creates an n-tromino from 1-tromino using as few calls to the genie as you can.

Solution
Pseudo-Code:

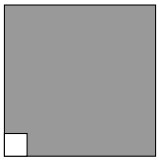
```
def TROMINO(n):
        if n == 1:
            return (2*2-1)
        if n == 2:
            return GLUE(GLUE(
                GLUE(DUPLICATE(TROMINO(n-1)), DUPLICATE(TROMINO(n-1))),
                GLUE(DUPLICATE(TROMINO(n-1)), DUPLICATE(TROMINO(n-1)))),
                DUPLICATE(TROMINO(1)))
        else:
            return GLUE(GLUE(
                GLUE(DUPLICATE(TROMINO(n-1)), DUPLICATE(TROMINO(n-1))),
                GLUE(DUPLICATE(TROMINO(n-1)), TROMINO(n-1))),
                DUPLICATE(TROMINO(1)))
```

Proof by Induction.

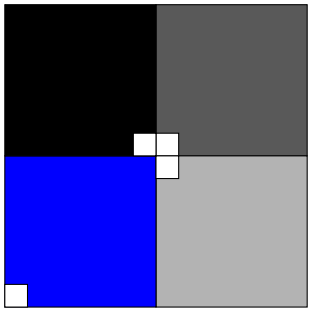
Proving that the algorithm shown above is most effective and hence, is the solution.

Base Case: For n =1, Tromino(1) itself is the solution

Induction Step: Assuming for n=k, the Tromino(k) is correctly built.
For n=k, the Tromino will appear as following with $2^k * 2^k$ sides with $1*1$ square missing:



If we glue 4 such Tromino block together in the with $2^{k+1} * 2^{k+1}$ with a 1x1 missing on the side and an L-shaped missing in the center which can be depicted in the following figure:



The remaining portion in the middle is a perfect fit for a Tromino(1). Thus, adding that will complete a $2^{k+1} * 2^{k+1}$ missing with a 1x1 square i.e. a Tromino(k+1)

Therefore, $Tromino(k+1) = 4(Tromino(k)) + Tromino(1)$ □

Problem 1.b. Give a recurrence relation for the number of calls $T(n)$ to the genie, and solve it.

Solution. In the pseudo code there are 4 calls for both `GLUE` and `DUPLICATE`.

Thus,

$T(n) = T(n-1) + 8$, where $T(n)$ is referring to the number of genie calls.

On further expanding T(n-1) and continuing the process, we get:

$T(n) = T(2) + 8 * (n-2)$

$= T(1) + (9) + 8 * (n-2)$ (1 extra call in case of 2, to take care of 1-Tromino)

$= 0 + 9 + 8n - 16$ (as there are no genie calls for $T(1)) = 8n - 7$

$\therefore$ the Number of Genie calls $= 8n - 7$

□

Problem 1.c. You now have a $2^n * 2^n$ chocolate bar that you need to distribute among the faculty members. The head of the department, in a quintessentially eccentric way, wants to eat only one particular (arbitrary) square. The rest of the faculty dont care which squares they get, except that they each want 3 squares in an L-shape. By some coincidence, the number of squares is exactly enough for the number of faculty members. Prove that there is some way to break up the chocolate bar, for any choice of n and the department heads square.

Solution

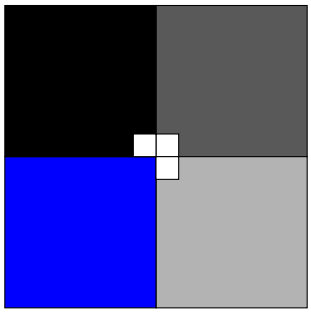
Base Case: For n =1, we get an L-shaped piece irrespective of whichever 1x1 square is chosen by the head of the department(HOD) which can be represented here:

Blue piece chosen by HOD.
Remaining is an L-shaped comprising of 3 squares.

Induction Step:
We assume that a solution exists for $2^k * 2^k$ chocolate bar such that it can be split into k L-shaped pieces of 3 squares and a single piece chosen by the HOD.

Now we prove that $2^{k+1} * 2^{k+1}$ can be split up in such a manner.

First we split the into 4 blocks after choosing HOD's piece in the following manner:



Now each of the blocks are of $2^k * 2^k$ size.

The HOD can select a piece randomly, let's say he wants a piece from the $2^(k) * 2^k$ blue block.

We know from induction that the blue square block with side $2^k * 2^k$ can provide such a split of a 1x1 piece and L-shaped pieces only utilizing all of the chocolate.

And the rest of the remaining blocks with shades of gray can also provide with such a split. But we don't need anymore 1x1 pieces. Thus, we can choose them to be the corner pieces for the $2^k * 2^k$ gray blocks but the center pieces for the $2^{k+1} * 2^{k+1}$ so that they form an L-shape piece comprising of 3 chocolate squares.

By induction, we know that after these 1x1 pieces from the center, each of the gray blocks can provide L-shaped pieces only utilizing all of the chocolate.

Thus, it's true for $2^{k+1} * 2^{k+1}$ chocolate bar to produce 1 random piece of chocolate for the HOD and rest in L-shapes for the rest of the faculty.

Problem 2.a. Prove that if A is rotation-sorted, then one of $A[0...(n/2-1)]$ and $A[n/2...(n-1)]$ is fully sorted (and, hence, also rotation-sorted with $c = 0$), while the other is at least rotation-sorted. What determines which one of the two halves is sorted? Under what condition *both halves* of A are sorted?

Solution.
Given: $A[i] = B[(i+c) \bmod n]$, where $B[0...(n-1)]$ is the sorted version of A and $i \in [0, n)$.

Let us consider the following cases:

Case 1: $c = 0$
$\implies A[i] = B[i \bmod n]$
$\therefore A[i] = B[i]$ (as $0 \leq i < n$)
It's given that B is sorted.
$\implies A$ is sorted as well here.

Case 2: $0 < c < \frac{n}{2}$
We know that $0 \leq (i + c) \bmod n < n$
Let us consider when: $0 \leq (i + c) < n$ (Assumption)
$\implies 0 \leq i \leq (\frac{n}{2} - 1)$
Let $j = (i + c)$
$\implies 0 < j < n$
Also, we know
$A[i] = B[(c + i) \bmod n]$
$\implies A[i] = B[c + i]$ (as we assumed $0 \leq (i + c) < n$)
Since $0 \leq (i + c) < n$ and the fact that B is sorted,
$\implies A[i]\ \forall i \in [0, \frac{n}{2} - 1]$ is sorted. (as $0 \leq (i + c) < n$)
$\therefore$ the first half of A is sorted in this case.

Case 3: $\frac{n}{2} \leq c < n$
This case is very similar to case 2.
We know that $0 \leq (i + c) \bmod n < n$
Let us consider when: $0 \leq (i + c) < n$ (Assumption)
$\implies \frac{n}{2} \leq i \leq (n - 1)$
Let $j = (i + c)$
$\implies 0 < j < n$
Also, we know
$A[i] = B[(c + i) \bmod n]$
$\implies A[i] = B[c + i]$ (as we assumed $0 \leq (i + c) < n$)
Since $0 \leq (i + c) < n$ and the fact that B is sorted,
$\implies A[i]\ \forall i \in [\frac{n}{2}, n - 1]$ is sorted (as $0 \leq (i + c) < n$)
$\therefore$ the second half of A is sorted in this case.

Therefore, from case 1,2 and 3, we have proved that either one of the halves of A is sorted or the whole array is sorted(when $c = 0$).

Also, the value of c determines which half is sorted or if the complete array is sorted.

$\therefore$ If $c = 0$, complete A is sorted
else if $c \in (0, \frac{n}{2}) \implies$ the first half of A is sorted
else the second half of A is sorted.

□

Problem 2.b. Assume again that A is rotation-sorted, but you are not given the cyclic shift c. Design a divide-and-conquer algorithm to compute the minimum of A (i.e., B[0]). Carefully prove the correctness of your algorithm, write the recurrence equation for its running time, and solve it. Is it better than the trivial $O(n)$ algorithm?

Solution

Pseudo-Code:

```
def findMin(A, l, r):
    if r < l:
        return A[0]
    if r == l:
        return A[l]

    # python3 division else use int typecasting
    m = int((l + r)/2)

    if m < r and A[m+1] < A[m]:
        return A[m+1]
    if m > l and A[m] < A[m - 1]:
        return A[m]
    if A[r] > A[m]:
        return findMin(A, l, m-1)
    return findMin(A, m+1, r)
```

Correctness:

Base case: For n=1, the only element is the minimum element.

Induction Step:

Assumption : For k elements, $findMin$ finds the solution i.e, the minimum element.

Now for k+1 elements, the algorithm checks if the middle element is greater than the right most element, then the smallest element must be to the right. If not, the smallest element must be to the left.

Thus, it checks one of the halves with floor((k+1)/2) elements [the one with the smallest element] for which (such sizes) we know it finds the minimum element. Thus the problem is reduced to half the number of elements. □

Running Time:

At each iteration, we decrease the input into half, thus we should arrive at O(log n) running time intuitively.

Recurrence Equation:

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

For a =1, b=2 and f(n) = constant
$\implies f(n) = \Theta(n^{\log_2 1})$
$f(n) = \Theta(1)$
Using Master's theorem, we get:
$T(n) = \Theta(\log(n))$
Thus, this is obviously better than the trivial $O(n)$ algorithm.

□

Problem 3. Describe an $O(\log n)$ divide-and-conquer algorithm to find some local minimum of a given (unsorted) array A of size n. You must give pseudo-code, prove correctness, and prove the running-time bound to get full credit.

Solution

An efficient solution is based on Binary Search. We compare middle element with its neighbors. If middle element is not greater than any of its neighbors, then we return it. If the middle element is greater than its left neighbor, then there is always a local minimum in left half. If the middle element is greater than its right neighbor, then there is always a local minimum in right half.

Pseudo-Code:

```
def localMinUtil(arr, low, high):
    n = len(arr)
    # python3 division else use int typecasting
    mid = low + (high-low)//2

    if ((mid == 0 or arr[mid-1] > arr[mid])
    and (mid == n-1 or arr[mid] < arr[mid+1])):
        return mid

    elif(mid > 0 and arr[mid-1] < arr[mid]):
        return localMinUtil(arr, low, mid-1)

    return localMinUtil(arr, mid+1, high)

def localMin(arr):
    return arr[localMinUtil(arr, 0, len(arr)-1)]
```

Correctness:

Base case: For n=1, the only element is the minimum element which is the local as well as global minimum.

Induction Step:

Assumption : For k elements, $localMin$ finds the solution i.e, a local minimum element.

Now for k+1 elements, the algorithm checks if the middle element is greater than the right element, then a local minimum must be to the right. If not, a local minimum must be to the left. This is because in either cases mid element is greater which implies the other neighbor of the mid's neighbor must be equal, less or greater than.

In the case of equality and less than, as we already have a greater than element which is the mid, there must be either a greater element on right side of the mid or all the elements on the right side of mid will be in non-ascending order. If they are in a non-ascending order, the corner-most element must be a local minimum.

In the case of greater than, then simply the mid's neighbor in question is a local minimum.

Thus, it checks and appropriately chooses one of the halves with floor((k+1)/2) elements and having a local minimum, for which (such sizes) we know the algorithm correctly finds a local minimum element. Thus the problem is reduced to half the number of elements. □

Running Time:

At each iteration, we decrease the input into half, thus we should arrive at O(log n) running time intuitively.

Recurrence Equation:
$T(n) = T(\frac{n}{2}) + Constant$
For a =1, b=2 and f(n) = constant
$\implies f(n) = \Theta(n^{\log_2 1})$
$f(n) = \Theta(1)$
Using Master's theorem, we get:
$T(n) = \Theta(\log(n))$

□

Problem 4. You are given a single-peaked array, which is an array $A[1, ..., n]$ such that there exists an index i such that $A[1, ..., i]$ is sorted in ascending order and $A[i + 1, ..., n]$ is sorted in descending order. You can assume that A contains distinct numbers. Design a divide-and-conquer algorithm to find the index of the maximum element in $O(\log n)$ time. Again, you must give pseudo-code, prove correctness, and prove the running-time bound to get full credit.

Solution

An efficient solution is based on Binary Search. We compare middle element with its neighbors. If middle element is greater than both of its neighbors, then we return it. If the

middle element is greater than its left neighbor and smaller than its right neighbor, then the max element is in right half. If the middle element is greater than its right neighbor and smaller than its left neighbor, the max element is in left half.

Pseudo-Code:

```
def findMaximum(arr, low, high):
    if (low == high):
      return arr[low]
    if ((high == low + 1) and arr[low] >= arr[high]):
       return arr[low]

    if ((high == low + 1) and arr[low] < arr[high]):
       return arr[high]

    # python3 division else use int typecasting
    mid = (low + high)//2

    if ( arr[mid] > arr[mid + 1] and arr[mid] > arr[mid - 1]):
       return arr[mid]

    if (arr[mid] > arr[mid + 1] and arr[mid] < arr[mid - 1]):
      return findMaximum(arr, low, mid-1)
    else:
      return findMaximum(arr, mid + 1, high)
```

Correctness:

Base case: For n=1, the only element is the maximum element.

Induction Step:
Assumption : For k elements, $findMaximum$ finds the solution i.e, the maximum element.

Now for k+1 elements, the algorithm checks if the middle element is greater both its neighbor, then mid is the max. Else checks if mid is greater than the right element and smaller than the left element, then the maximum must be to the left. If not, a local minimum must be to the right. This is because it's given that the array is a peaking array.

Thus, it checks and appropriately chooses one of the halves with floor((k+1)/2) elements and having the maximum for which (such sizes) we know it finds the maximum element. Thus the problem is reduced to half the number of elements. □

Running Time:

At each iteration, we decrease the input into half, thus we should arrive at O(log n) running time intuitively.

Recurrence Equation:
$T(n) = T(\frac{n}{2}) + Constant$
For a =1, b=2 and f(n) = constant
$\implies f(n) = \Theta(n^{\log_2 1})$
$f(n) = \Theta(1)$
Using Master's theorem, we get:
$T(n) = \Theta(\log(n))$

□