Fundamental Algorithms

Homework 1

Tushar Jain, N12753339

Q1. (Induction)

(a) *To Prove*:

$$S(n) = 1^3 + 2^3 + \ldots + n^3 = (1 + 2 + \ldots + n)^2 = [\, n (n + 1) / 2]^2 \qquad \text{[eq. 1]}$$

Base Case, k = 1:

$$S(1) = 1^3 = [1 * (1 + 1) /2]^2 = 1$$

Induction Step: We assume $S(k) = (1 + 2 + \ldots. + k)^2$ to be true.

We must now show it's true for (k+1)th step as well.

$$\Rightarrow S(k+1) = S(k) + (k+1)^3$$

$$= [k(k+1)/2]^2 + (k+1)^3$$

$$= \frac{k^2 (k+1)^2}{4} + (k+1)^3$$

$$= \frac{k^2 (k+1)^2 + 4 (k+1)^3}{4}$$

$$= \frac{(k+1)^2 [k^2 + 4k + 4]}{4} \quad \text{[taking } (k+1)^2 \text{ common]}$$

$$= \frac{(k+1)^2 (k+2)^2}{4}$$

$$= \frac{[(k+1)\ (k+2)]^2}{2^2}$$

$$= [\, 1 + 2 + \ldots + k + (k + 1)]^2$$

This shows that S(k) implies S(k+1). Hence, it's proved [eq.1] holds for all k belonging to a natural number.

(b) *To Prove:* the product of arbitrarily many odd numbers $x_1, x_2, \ldots, x_n$ is itself odd.

Let the number of odd numbers be m. And on the basis that we can represent any odd number as (2n + 1) where $n \in \mathbb{Z}$.

Base Case, m = 2: For $x_1 = (2a + 1)$, $x_2 = (2b + 1)$ where $a,b \in \mathbb{Z}$, that is,

$$\begin{aligned}(2a+1)(2b+1) &= 4ab+2a+2b+1 \\ &= 2(2ab+a+b)+1\end{aligned}$$

, where $2ab+a+b \in \mathbb{Z}$, so it is true for $m=2$.

By *induction*, we assume for arbitrary $k \in \mathbb{Z}$ that,

$$\prod_{i=1}^{k}(2a_i + 1) = 2l + 1 \text{ where } l \in \mathbb{Z}.$$

Since by our basis step we have that two odd numbers multiplied together gives two odd numbers, then for the (k+1)th odd integer *c*, we have that

$$(2l + 1)(2c + 1) = 2(2lc + l + c) + 1$$

So, since $(2lc + l + c) \in \mathbb{Z}$, we have that

$$\prod_{i=1}^{k+1}(2a_i + 1) = 2j + 1 \text{ where } j \in \mathbb{Z}.$$

Therefore, by the Principle of Mathematical Induction, the product of any m odd integers is again odd.

(c) *To Prove:* The minimum number of breaks needed to split a $m \times n$ grid of chocolates into 1×1 pieces is $mn - 1$.

Intuitive trick: To break up a big chocolate bar, we need one split to make two pieces, & then we can break up the two pieces recursively. This suggests a proof using strong induction on the size of the chocolate bar, where size is measured in chocolate squares. Now instead of a problem involving two variables (the two dimensions), we have a problem in one variable (the size). With this simplification, we can prove the theorem using strong induction.

Let $P(k)$ be the proposition that a chocolate bar of size k requires at most $k - 1$ splits.

Base case, k = 1:

$P(1)$ is true because there is only a single square of chocolate, and $1 - 1 = 0$ splits are required.

Induction step:
Assuming $k \geq 1$ and any chocolate bar of size s, where $1 \leq s \leq k$, requires at most $s - 1$ splits.

Now we must show there is a way to split a chocolate bar of size k + 1 with at most k splits.

We do this by first breaking the chocolate bar of size k + 1 into two smaller pieces of size p and q where p + q = k + 1. This is certainly possible because the size (k) of the bar more than one.

Now the pieces of sizes p and q are between one and k, so by strong induction, breaking these two pieces into single squares requires only p − 1 and q − 1 splits, respectively.

Thus, the total number of splits required to break the bar of size k + 1 into single squares (that is 1×1) is therefore at most

$$1 + (p - 1) + (q - 1) = p + q - 1 = (k + 1) - 1 = k.$$

This shows that P(k) implies P(k + 1), and the claim is proved by strong induction.

Q2. Median in an unsorted array of length n

(a) We use *pivots* and *partitions* in a similar manner to how they are used in *quicksort*. Pivots are the elements chosen during the process of partition such that all elements less than the pivot are on 1 side and all the other elements ($\geq$ pivot) on the other side. We simply recursively call the partition method again and again till the time the pivot reaches the $(n/2 + 1)th$ element.

(b) Let A be the array with length given as n and assuming 1 indexing of array. Following CLRS pseudo-code notation, here is the pseudo code for the same.

RANDOMIZED-PARTITION (A, p, r)

```
x=A[ RANDOM(p,r) ]
i=p - 1
for j=p to r - 1:
        if A[j] <=  x
                i=i+1
                exchange A[i]   with A[j]
exchange A[i+1]   with A[r]
return i+1
```

RANDOMIZED-SELECT(A,p,r,i)

```
if p == r:
        return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k= q - p + 1
if i == k // the pivot value is the answer
        return A[q]
else if i<k
        return RANDOMIZED-SELECT(A,p,q - 1,i)
else return RANDOMIZED-SELECT(A,q+1,r,i -k)
```

MEDIAN(A,n)

```
return RANDOMIZED-SELECT(A,1,n, FLOOR((n+1)/2))
```

(c) Assuming writing and reading an array element both cost $1.

Also, let's define exchange as following:
EXCHANGE(a,b) // call by reference

temp = a
a = b
b = temp

Thus, the cost of exchanges would be 3$ (assuming code in this function (and NOT the function itself as using the function would make 2 more read calls cost $2 extra) is used at all places in pseudo code where the exchange with is mentioned).

The worst case for this algorithm would be when we're extremely unlucky and the partition occurs over the largest remaining element.

Now we calculate the cost of the partition function in the worst case (pivot is the largest number & thus if is true every time).

=> worst case partition cost = $ $(1 + (r - 1 - p) * (1 + 3) + 3)$

= $ $4(r - p)$

In the worst case, the RANDOMISED-SELECT would call partition n - 1 times (or r-p-1) and would read an array element while returning the value

=> worst case selection cost = $ [($r - p - 1$) * partition cost +1]

Thus for finding the median of array of length n the cost is as follows (that is r = n, p = 1) :

=> Worst Case Median Cost = $[$(n - 1 - 1) * 4(n - 1) + 1]$

= $ $[4(n - 1)(n - 2) + 1]$

Q3. Beach Problem

(a) This problem can be easily solved by simply finding the first arrival value (i.e. minimum of a[]) and the last departure value(i.e. the maximum of b[]) and we must stay at the beach for this duration to meet optimal solution requirements.

(b) Let a[] and b[] be the given arrays each of length n and assuming 1 indexing of array.

Following CLRS pseudo-code notation, here is the pseudo code for the same.

```
MINIMUM(A)
        min = A[1]
        for i=2 to A.length
                if min > A[i]
                min = A[i]
        return min

MAXIMUM(A)
        max = A[1]
        for i=2 to A.length
                if max < A[i]
                max = A[i]
        return max

SOLUTION(a,b)
        arrival = MIN(a)
        departure = MAX(b)
        display arrival and departure time
        return departure - arrival
```

(c) Assuming each array of length n (as there are n friends).
Number of comparisons for both minimum and maximum function is $n-1$
Thus total cost of algorithm is $\$ 2(n-1)$.

Q4. Transpositions and Insert-Sort

(a) There are 5 transpositions in the array: [2, 3, 8, 6, 1] which are as follows

(2, 1) (3, 1) (8, 6) (8, 1) (6, 1)

(b) The reverse sorted n-element array [n, n-1, …,2, 1] has the most Transpositions. It has the following number of transpositions:

$$n-1 + n-2 + \ldots + 2 + 1 = \frac{n(n-1)}{2}.$$

(c)

INSERTION-SORT(A)

```
for j = 2 to A.length
        key=A[j]
        // Insert A[j]   into the sorted sequence A[1 …  j  - 1] .
        i=j-1
        while i >0 and A[i] >key
                A[i+1] = A[i]
                i=i -1
        A[i+1] = key
```

The running time of insertion sort is proportional to the number of inversions in the set. If $I(i)$ denotes the number of $j < i$ s.t. $A[j] > A[i]$.

=> Number of inversions in $A = \sum_{i=1}^{n} I(i)$

The while loop in the insertion sort algorithm executes once for each element of A which has index less than j and is $greater\ than\ A[j]$. Thus, it will execute $I(j)$ times. We reach this while loop once for each iteration of the for loop, so the number of constant time steps of insertion sort is $\sum_{j=1}^{n} I(j)$ which is exactly the inversion number of A.

The running time of the inner loop is proportional to the number of inversions in the set and therefore the running time of insertion sort (ignoring individual instruction time) can be expressed as (n * numberOfInversions). This is because the insertion sort's inner swapping loop works by swapping inversions.