COMP3121 Assignment 1

Alex Nguyen

20/03/16

Question 1

Generally, the thieves should extend the strategy adopted by 3 thief scenario. To start, the first thief should take what he believes is from the total. Now check with the others if they also believe that this is the right amount.

If any disagree, recombine the items and allow the thief who disagreed to split the pile, again checking for the agreement of other thieves. Repeat until all thieves have taken $\frac{1}{n}$. Code written in pseudo-C type language is given below to demonstrate:

```
int main (int argc, char * argv[]) {
    int pileSize;  //this is the total pot; reduce as thieves take
    int n;  //number of thieves
    int *thiefArray = createNewArray(n);
    //thiefArray[i] records whether thief at each index
    // has taken from the pot. each initialised to 'no' below
    initialise(thiefArray);
    int *spoilsArray = createNewArray(n);
    //spoilsArray will contains the winnings of the i'th thief

    // Number all thieves; let the current thief be the i'th thief
    for (int i = 0; i < n; i++) {
        split (thiefArray, spoilsArray, i, pileSize);
    }
    //Print out the spoils of each thief
    for (int i = 0; i < n; i++) {
      printf ("Thief %d took %d things", i, spoilsArray[i]);
    }
}
//curThief is index of thiefArray to take from pile
void split (thiefArray, spoilsArray, curThief, pileSize) {
    //Compute fair share according to curThief
    valTaken = pileSize/n;
    //Check with all other thieves(j) that curThief's split is fair
    for (j = 0; j < n; j++) {
        next if (thiefArray[j] != 'no' or j == curThief);
        //thiefArray[j] == no:j has already taken his share - no say
        //Thief i obviously thinks it's fair, exclude him from check
        //Ask remaining thieves if this amt is fair
        if (!agree(j, valTaken))) { //defined below
            //If a thief j disagrees:
            //The pile is recombined, now THIS thief j attempts
            //to take his share.
            split (thiefArray, spoilsArray, j, pileSize);
        }
    //continued over page
```

```
        //outside check: take from the pot (with others permission)
        spoilsArray[i] = valTaken;
        //effectively deletes this thief from consideration
        thiefArray[i] = 'yes';
        //past check; all other thieves agree this is 1/n of share
        *pileSize -= valTaken; //so thief i reduces pileSize
        //remove thief and move positions of all, other
        // thieves left; he has taken his share
    }
}
    bool agree(thiefNo, value);
    // INPUT: takes in value which might be taken by
    // current thief and another thief in line so he can check the size
    // OUTPUT: returns true/false whether checked thief
    // Agrees that value = pileSize/n
    // Not implemented as this is subjective to each thief
```

Question 2

Problem Statement:

- In a room of 20 people (including himself) Tom got 19 different answers from other people (wife included) (1)

- Nobody shook hands with themselves (2)

- Nobody shook hands with their spouse (3)

To show the number of hands Tom shook, first show that Mary shook 9 hands.

Combining (1) with (2) and (3), of the group the highest number of handshakes for an individual was 18 (-1 for each condition). (Someone knew nobody)

From 1), someone also knew everyone and shook 0 hands, as range 0..18 gives 19 unique shake combinations

- The person who shook 0 hands (P0) is married to the one who shook 18 (P18). (Everyone BUT P18's wife shook hands with him. So, P0 must be the only person who didn't - his wife)

Remove P18 and P0 from the available handshake numbers, leaving 1..17 handshakes from the rest. However, removing P18 also removes him from having shaken the hands of the others. Hence, the range is shifted by 1 ('New' handshakes)

Consider the below table which extrapolates the steps:

Remaining Couples	Range Total Handshakes	Range 'New' Handshakes
9	0..18	0..18
8	1..17	0..16
7	2..16	0..14
...	...	...
2	7..11	0..4
1	8..10	0..2

Now, Mary cannot be removed – Tom is not part of the picture and handshakes are tied to partners to have tihs pattern. Hence, removing the last couple will remove the '8' and '10' entries from the last table row, leaving Mary with 9 handshakes.

From table, we can also see relationship between spouse to create 19 unique answers (n is handshakes):

$$n_{\text{person}} = (n_{\text{total}} - 1) - n_{\text{spouse}} \tag{1}$$

Since Tom was Mary's spouse and participating in the handshakes, he contributed to the handshake range above. For Tom:

$$n_{\text{Tom}} = 19 - 1 - n_{\text{Mary}} n_{\text{Tom}} = 9 \tag{2}$$

So, Tom shook 9 hands also.

Question 3

In finding a path, always find one to taking the first 'turn'/ intersection. If this is not possible without changing direction (going against street sign), then there we say there is no path. (pathExists is false below) Define circumnavigating blocks by being able to reach the same point by following all street signs.

Algorithm: To find a block, start with outer circumference highway and confirm that 1 block exists (inner circle) by observing that a path exists (according to above definition). Then, add a unidirectional street and confirm that a block exists by checking whether there is a path back to itself for every intersection. Continue this for n roads and return on the final road with the road. This is described by the below pseudocode:

```
initialise empty highway graph
failString = ``Could not find road''
//first road is outer circle
for each currentRoad
do
    add road to highway graph
    for each intersection on highway
    do
        if pathExistsBackToMe(currentIntersection)
        //Path is with first possible intersection always taken
        //without disobeying street signs (change direction)
            if currentRoad = last road
                return currentRoad
            end if
        end if
    done
done
//if reached and not returned, path was not found
//no block obeying all street directions exists
    return failString
```

Question 4

Consider situations from n = 1 to n = 5 pirates.

n = 2: P4 P5

Say, P4 proposes first. It does not matter what P4 proposes as P5's optimal move will be to reject (and effect a tie) to kill P4 for fun and take all the gold.

Here spoils are: P4: 0; P5: 100 bars

n = 3: P3 P4 P5

Now, P3 proposes first. As pirates are good problem solvers, P3 knows that P4 will face the result from n = 2 if he dies. As P4 does not want to die above all else, he will agree with P3 regardless (and win majority vote) - P3 does not have to distribute any as P5 will be outvoted 2-1.

Here spoils are: P3: 100 bars; P4: 0; P5: 0

n = 4: P2 P3 P4 P5

Now, if P2 dies then the n = 3 situation will arise. As P3 wins in that situation, he will reject P2's proposition regardless. P4 and P5 stand to gain nothing from the n = 3 scenario and will reject P2 for fun. Thus, P2 should buy votes from P4 and P5 with 1 bar each to win the vote.

Here spoils are: P2: 98 bars; P3: 0; P4: 1; P5: 1

n = 5: P1 P2 P3 P4 P5

If P1 dies, then n = 4 will arise. P2 knows the strategy to win that so will reject P1 regardless. Again, P3 and P4 will reject P1 to kill him for fun unless P1 offers him more than the alternative (n = 4 scenario). If P1 can win P3 and P4 over, P5's vote does not matter since P1 will have majority. P1 should beat the spoils of n = 4, giving P3 1 bar and P4 2 bars respectively.

Here spoils are (which should be P1's proposition): P1: 97 bars; P2: 0; P3: 1; P4: 2; P5: 0

Question 5

a) Sorting the array can be done in $O(n \log n)$ - complete by calling Mergesort. For every array element i, the corresponding element needed to sum to x can then be found by searching for x - S[i].

Binary search on an array is also done in $log_2 n$ time, repeated over an array of n terms yields an overall $O(n \log n)$ complexity. A pseudocode implementation in C-like syntax is given below:

```
boolean findSum (int* S, int x) {
    n <- length[S]
    MergeSort(S,0,n)
    //Array is now sorted. MergeSort has O(nlogn)
    for i = 0 to n do
        remainder = x - S[i]
        return true if findInArray(S, remainder, 0, n) //bool returned
    end for
    return false
}

boolean findInArray(int *array, int val, int lo, int hi)
{
    if hi = lo && a[lo] = val  //perhaps check it's in range of array
        return true
    else
        return false

    mid = (lo + hi) /2
    if array[mid] > val findInArray(array, val, lo, mid)
    else if array[mid] < val findInArray(array, val, mid +1, hi)
    else return true //if array[mid] == val
}
```

b) To achieve $O(n)$ complexity, using a hash table to store values that have occurred will keep track of value encountered whilst scanning through an array. Scan through and add every element to the hash table to signify it has been encountered. Then, with every element check whether the 'remainder' (x - S[i]) exists within the hash table.

If the sum exists, the matching entry will have been encountered in the hash table by the end of the array. As hashes have $O(1)$ searching complexity, the overall complexity will be from array scanning: $O(n)$. See pseudocode below:

```
boolean findSum (int* S, int x) {
    n <- length[S]
    h = createHashTable(n) //Hash table with n slots
    for i = 0 to n do
        add (h, S[i]) //add current val to h
        remainder = x - S[i]
        //check if remainder has been encountered - if so,
        //a value exists to sum to x
        if (search(h, remainder)) return true
    end for
    return false
}
Here, search(h, val), add(h, val) and createHashTable(n)
are hash table data structure functions
```

Question 6

Consider how RadixSort works to use a sort optimised for a small range of values and applying them on individual digits (from least significant):

```
Radix-Sort(A,d)
    for i = 1 to d
        use a sort to sort A on digit i
```

For an array of n digits with values sized up to n^9, this can be considered as sorting n base n numbers with less than 9 digits. Then, the complexity is governed by the algorithm that sorts each digit.

CountingSort operates on $O(9(n+k))$, where k scales on the range of value of the digits which overall yields $O(n)$ complexity Counting uses the cumulative frequency of each digit in order to sort rather than comparisons, allowing its relative speed compared to comparative ones like MergeSort.

```
Radix-Sort(A,9)
    for i = 1 to 9 //operates on 1 element of each
                   //array entry simultaneously - O(n)
        CountingSort(A)
    done
```

Question 7

For $x_1..x_n$ in the interval $[0,1]$, sorting yields $y_1 = 0$ and $y_n = 1$. Hence, (sum is $i = 2$ to n)

$$\sum |y_i - y_{\text{i-1}}| = y_2 - y_1 + y_3 - y2 + .. + y_n - y_{\text{n-1}} = y_n - y_1 \tag{3}$$

From above, $y_n - y_1 = 1 - 0 < 2$.

Thus, on a sorted array equation $\sum |y_i - y_{\text{i-1}}| < 2$ holds true. To accomplish this in linear time $O(n)$ consider the BucketSort Algorithm pseudocode:

```
BucketSort(A)
  let B[0..n-1] be a new array
  n = A.length
  for i = 0 to n - 1
    make B[i] an empty list
  for i = 0 to n
    insert A[i] into list B[nA[i]]
  for i = 0 to n - 1
    sort list B[i] with insertion sort
  concatenate lists B[0], B[1] \dots B[n-1] together in order
```

The above has $O(n^2)$ which simplifies to $O(n)$ for small numbers of elements, which is not sufficient in our general case. This is due to the insertion sort which sorts elements placed in buckets - slows down as more elements are BucketSorted.

However, if the number of buckets (essentially list array B) was equal to the number of possible values being passed in; i.e. $B[0..N]$ where N is range of values in A - there would be one bucket for every input value which eliminates the need for insertion sort.

Thus, the concatenation step would naturally sort the values and result in a linear $O(n)$ BucketSort of A to make equation (3) ¡ 2 true.

Question 8

As shoes and feet cannot be directly compared, quicksort values based on pivots from the opposite array in one big array with two pivots. Shoes can be used to compare too big or too small and vice versa. Repeating for all shoes and feet will result in sorted sub-arrays of shoes and feet. The kid's foot can easily be matched from lowest size to highest by popping off the start of each sub-array one at a time.

The below is a modified quicksort pseudo-code to demonstrate the idea:

```
combine shoe size and foot array into A
save ends of each array
swap arbitrary foot and shoe into either array
save their index
call quicksort(A, 1, lengthShoes, lengthShoes+1, A.length)
swap foot and shoe back into original index (?)
split final array back up to shoe and feet.
Scan through from low to high and A1 and A2 index i values will fit

//p, r - start or end of shoes; x, z start end feet
quicksort(A, p, r, x, z)
    if p < r
        exchange A[r] with A[z]
        q = partition(A, p, r, z)
        quicksort(A, p, q - 1, x, z) //lower half of shoe array
        quicksort(A, q + 1, r, x, z) // top half of shoe array
    else if x < z
        exchange A[r] with A[z]
        y = partition (A, x, z, r)
        quicksort(A, p, r, x, y - 1) //lower half of feet array
        quicksort(A, p, r, y + 1,  z) // top half of feet array


//x is the pivot at r -- somehow swap a shoe for foot in each array
    //before call here.
partition(A, p, r, z)
    x = A[r]
    i = p -1
    for j = p to r - 1
        if sizeCompare(A[j], x) // Compares pivot foot to shoe
        //returns boolean depending if too big or too small.
        //too big => size < A[r] too small => size > A[r]
            i = i + 1
            exchange A[i] with A[j]
    exchange A[r] with A[z] //partitioned, restore to array section of
                            //shoes/feet separately
    exchange A[i + 1] with A[r]
    return i + 1
```

Question 9

Assuming both arrays are equally sized and an $O(nlog_2n)$ search used to prep B, this can be accomplished by using a binary search (like in Question 5) to check if an element from array A exists in array B. As this must be done for all elements in A, there is $O(n)$ which is multiplied by $O(log_2n)$ from the binary search dividing search array by 2 depending on whether the value is higher than the midpoint or not to (combine with sort complexity and) give $O(nlog_2n)$.

```
boolean findCommon (int* A, int* B) {
    n <- length[A]
    MergeSort(B,0,n)
    //Array is now sorted to prep for binary search.
    //MergeSort has O(nlogn)
    for i = 0 to n do //n time - scan through A
      return true if findInArray(B, A[i], 0, n)
      //match found!
      //log2n time - binary search used in findInArray
    end for
    return false
}

boolean findInArray(int *array, int val, int lo, int hi)
{
    if hi = lo && a[lo] = val
        return true
    else
        return false

    mid = (lo + hi) /2
    if array[mid] > val findInArray(array, val, lo, mid)
    else if array[mid] < val findInArray(array, val, mid +1, hi)
    else return true //if array[mid] == val
}
```

Question 10

Algorithm strategy: Initialise a hash table h.

Scan through the A[1..100]'s n = 100 items and search hash table for A[i] as it is encountered. add items to h if searchHashTable returns false. If search is successful on encounter, then it has been encountered before (and thus is the duplicate).

As h has $O(1)$ accessing and searching complexity, the dominant complexity is $O(n)$ as all entries of A need to be traversed.

```
boolean findDup (int* A) {
    n <- length[S]
    h = createHashTable(n) //Hash table with n slots
    for i = 0 to n do
        if (search(h, A[i])) return A[i]
        //Search hash for A[i] - if true, then previously
        //encountered - this is the duplicate
        add (h, S[i]) //add current val to h to compare
    end for
    return false
}
Here, search(h, val), add(h, val) and createHashTable(n)
are hash table data structure functions
```

Question 11

Petrol stations in total have enough for a round trip. Hence, at least 1 station has enough to get to the next. Let $d_i = p_{\text{station}} - p_{\text{usedToStation}}$ At any station the sum of all d to a station is: $s_i = d_1 + d_2 + \ldots d_i$

Let d_k be the station with the minimum d value (assuming negative fuel is possible)

So, $s_k = d_1 + .. + d_k$

Since k had the minimum difference, from the k'th station to the last station numericallythe sum is then:

$s_j - s_k \geq 0$, noting that $s_{=}d_1 + .. + dj$ and $k < j \leq n$

Extending back around to the k'th station we have: $1 \leq j \leq k$

$s_j = d_1 + .. + d_k$ and $s_n = d_1 + .. + d_k + .. + d_n \geq 0$

We have travelled to all stations so: $d_1 + .. + d_k + .. + d_n$

But $s_n - s_k = d_k + ..d_n$ and $s_j = d_1 + .. + d_k$ Therefore after all stations travelled amount in tank is: $s_n - s_k + s_j = s_n + (s_j - s_k) \geq 0$

Thus, picking one minimum station has resulted in a ≥ 0 net petrol in the car after a round trip.

Question 12

a) $f(n) = n$ and $g(n) = (n2log_2n)(n + cosn)$: $f(n) = \Omega(g(n))$ for $n > 4$ and $c \geq 5$

a) $f(n) = (log2n)^2$ and $g(n) = log2(nlog2n) + 2log2n$: $f(n) = O(g(n))$ for $n > 1$ and $c \geq 1$

Question 13

For an array of size 2^n, let N be the number of items. We require $3(2^{n-1}) - 2$ comparisons. In terms of N: $\frac{3N}{2} - 2$ In pseudocode below, there are 3 comparisons and values are considered 2 at a time:

```
void maxMin (int * max, int * min, int* A, int N)
    for (int i = 0; i < N; i += 2) {
        j = i + 1;
        //assume max and min are initially empty
        bool cmpResult = cmp(A[i], A[j]);
        //first comparison
        //Don't have to compare with max and min on first run
        if (*max == NULL && *min == NULL && cmpResult) {
                *max = A[i];
                *min = A[j];
        } else {
            *min = A[i];
            *max = A[j];
        }

        if (cmpResult) {
            if (cmp (A[i], *max)) { //second comparison
                *max = A[i];
            }
            if (cmp (min, A[j])) { //third comparison
                *min = A[j];
            }
        } else {
            if (cmp (A[j], *max)) { //see above
                *max = A[j];
            }
            if (cmp (min, A[i])) { //see above
                *min = A[i];
            }

int cmp (int a, b) {
```

```
    if a > b return true;
    else return false;
}
```

Further, since max and min don't have to be compared on first run through, this saves 2 iterations. The number of comparisons then comes to $\frac{3N}{2} - 2$ which as shown above is $3(2^{n-1}) - 2$.

Question 14

As in Question 13, $2^n + n - 2 = N - log_2(N) - 2$ The following pseudocode gives N comparisons to set max/second max values (see comments) and incorporates a binary search to give $log_2(N)$ complexity. As above, initialising values save 2 comparisons to give total $2^n + n - 2$ comparisons.

```
int max;
int max2;

void max2max (int * max, int * max2, int* A, int i, int j) {

    if (!max && !max2) { //if no value in maxes, set them to first values
      //save 2 comparisons
        *max = A[i]
        *max2 = A[j]
    }
    if (i <= j) return i; //binary search comparison
    mid = (i + j)/ 2;
    //2 comparisons 2 at a time - n comparisons
    if cmp (A[mid], *max) {
        if (cmp(*max, *max2)) {
            *max2 = *max;
        }
        *max = A[i];
         return max2max (max, max2, A, mid + 1, j);
    } else {
         return max2max (max, max2, A, i, mid);
    }
}
```

Question 15

From the question conditions, 2 things are true from asking if A knows B (A k B) If AkB then A can't be a celebrity. If not, then B can't be a celebrity. This can be done by asking everyone at the party sequentially in an array.

Say, n = 5 for $ABCDE$ Compare A and B first, ask A k B ? From above, at least one will be eliminated (say A) so that the people to ask become: $BCDE$

This is repeated until (n - 1) questions have been asked to have a final possible celebrity candidate. (Say, E in this case)

Their celebrity status is then confirmed by asking n attendees (n - 1) questions if they know the final candidate (A k E, B k E, etc).

Finally, ask the celebrity if they know everyone else (E k A, E k B etc) for a final (n - 1) questions to confirm he knows nobody else.

Summing up a celebrity presence will be confirmed in $3(n-1) = 3n - 3$ questions.