Homework Goals

• Confirm ability/competence in a C++ programming environment;
• Develop ability to work in ‘Mars Lander’ environment;
• Confirm ability to compile and execute in above environment(s), and specifically, in environment in which projects are due;
• Convert C++ code to pseudocode, and pseudocode to C++ code;
• Analyze complexity of algorithms using empirical analysis, solution to recurrence relations, and ‘observation’.
  • Plot performance results (i.e., do empirical analysis) for different input sizes;
  • Solve recurrence relations for various recursive algorithms;
  • Complete operation count analysis and simplify results for iterative algorithms;
• Use base data structures (lists and arrays) in the implementation of basic Abstract Data Types (stacks and queues);
• Demonstrate fundamental understanding of Hash Functions by applying Hash Codes and Compression Maps.

Problem 1: 15/50 points

Problem 1a) Programs for calculating prefix averages are given in the text for a quadratic time algorithm (code fragment 3.4, page 133) and a linear time algorithm (code fragment 3.5, page 134). Assume that the $X[0] = 0$, $X[1] = 1$, $X[2] = 2$, …

1. Convert the code from pseudocode to compilable C++ code.
2. Type, compile, and execute the programs in the C++ programming environment of student’s choice.
3. Compile and execute the same code on a CAEN Solaris machine (loginsun.engin.umich.edu).
4. Run each program on input ($n$) sizes of 10, 100, 1000, 10000, 30000, 70000, and 100000. Measure and record the execution time for all input sizes (see time manpage). A student may use ‘user CPU’ or ‘wall clock time’ for timing execution in environment two.
5. Plot execution time for various input sizes in the plotting tool of the student’s choice. Possibilities include: MS-Excel, Gnuplot.

Hints:
• The student may want to check output through screen I/O prior to timing actual results. However, screen output that is used for debugging should be removed prior to accumulating timing results.
Turn-ins for Problem 1
• Two column, line-by-line format showing actual C++ code vs. pseudocode given in book.

### Quadratic Pseudocode
Let A be an array of N numbers

```
for i<0 to n-1 do
    a<-0
    for j<0 to i do
        a<-a+X[j]
    A[i]<-a/(i+1)
```

### Quadratic C++
```
double* averages;
averages = new double[size];
for(int i = 0; i < size; i++)
{
    double a = 0;
    for(int j = 0; j < i; j++)
    {
        a = a + array[i];
    }
    averages[i] = a/(i+1);
}
return array A
```

### Linear Pseudocode
Let A be an array of n numbers

```
s<-0
for i<0 to n-1 do
    s<-s+X[i]
A[i]<-s/(i+1)
```

### Linear C++
```
double* averages;
averages = new double[size];
double s = 0;
for(int i = 0; i < size; i++)
{
    s = s + array[i];
    averages[i] = s/(i+1);
}
return averages;
```

• Sentence describing the environment used to develop solution to problem 2. Sentence describing the environment used to develop solution to problem 3. “Same” is adequate if the same environment was used for problems 2 and 3.
• The text of your program (*.c* file) used to implement the algorithms.
• A script (see script manpage) of the linear time program for input size n = 100000 being compiled, run, and timed in environment 3.
• A plot showing the execution times for the various input sizes listed in problem 4. The results for both the linear and quadratic time algorithms should be shown in the same plot, but no penalty if they are not (just mark it on the paper). No penalty if linear time shows little to no increase in execution time.

### Problem 2: 10/50 points
Develop a simple algorithm to move Vanguard from its current location to the nearby tile upon which is located a missing component (ATU, WTD, Crystal, Beagle 2). First, iterate over the Vanguard's current area looking for a missing component. After locating the item, create a path that moves (+/-) X tiles in the East/West direction and (+/-) Y tiles in the North/South direction. That is, if the Vanguard's current position is tile (3,3) and a missing item is located on tile (9,6), then Vanguard should move 6 spots in the East direction and 3 spots in the South direction.

Turn-ins for Problem 2
• Submit your code to the autograder
Problem 3: 10/50 points

3a) Convert the following algorithm from C++ to pseudocode using conventions described in the text.

```cpp
int lg (int n)
{
    for (int i = 0; n > 1; i++, n /= 2);
    return i;
}
```

Algorithm lg(n)
Input: integer n
Output: log₂(n)

```
i <- 0
while (n > 1) do
    i <- i + 1
    n <- n / 2
return i
```

3b) Complete an operation count analysis of the pseudocode algorithm derived above. Include the cost of all primitive operations in your analysis (e.g., variable assignment, arithmetic operation, …). That is, complete a ‘decomposition’ analysis of your algorithm.

1 assignment + (lg(n) + 1) comparisons + 2 lg(n) assignment + 2 lg(n) arithmetic + 1 return
= 5 lg(n) + 3 operations

arithmetic = 2 lg(n)
assignment = 2 lg(n) + 1
comparisons = lg (n) + 1
return = 1
Total = 5 lg(n) + 3
Problem 4: 5/50 points
Derive and solve the recurrence relation for the following recursive algorithm.

```c
int searchR(int a[], int v, int left, int right)
{
    if (left > right) return -1;
    int mid = (left + right)/2;
    if (v == a[mid]) return mid;
    if (v < a[mid])
        return searchR(a[], v, left, mid - 1);
    else
        return searchR(a[], v, mid + 1, right);
}
```

Recurrence Relation: \( T(n) = 1 + T(n/2), T(1) = 1, T(0) = 1 \)

Solution:
\[
T(n) = 1 + T(n/2)
\]
assume \( n = 2^x \) (and \( x = \log n \))
\[
T(2^x) = 1 + T(2^{x-1})
= 1 + 1 + T(2^{x-2})
\]
\[
= 1 + 1 + T(2^{x-2})
= 2 + T(2^{x-2})
\]
\[
= \ldots + 1 + 1 + T(2^0)
= \ldots + 1 + 1 + T(2^0)
= x + 1
\]
Plug in \( x = \log n \)
\[
T(n) = \log n + 1
\]

\( T(n) = O(\log n) \)
Problem 5: 10/50 points
In class we described the implementation of list-based stack and an array-based queue. Below, show the code for an array-based stack and a list-based queue.

Array-based stack

create(int size)
{
    A = new T[size];
    next = 0;
}

bool empty()
{
    return (next == 0);
}

bool full()
{
    return (next == size);
}

push(T elt)
{
    if (full())
        throw exception;
    A[next++] = elt;
}

T pop()
{
    if (empty())
        throw exception;
    return A[--next];
}

List-based queue

create()
{
    head = NULL;
    tail = NULL;
}

bool empty()
{
    return (head == NULL);
}

bool full()
{
    return false;
}

enqueue(T elt)
{
    node *n = new node;
    n->element = elt;
    n->next = NULL;
    if (tail)
        tail->next = n;
    tail = n;
    if (head == NULL)
        head = tail;
}

dequeue()
{
    if (empty())
        throw exception;
    T result= head->element;
    node *victim = head;
    head = victim->next;
    delete victim;
    if (head == NULL)
        tail = NULL;
    return result;
}
Use linear probing, a hash table with $b = 13$ buckets, and the hash function $f(k) = k \mod b$. Start with an empty hash table and insert items whose keys are 7, 42, 25, 70, 14, 38, 8, 21, 34, 11. Note that the items are inserted in the order given.

<table>
<thead>
<tr>
<th>$f(k)$</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
</tr>
</tbody>
</table>