Queensland University of Technology

CAB230 - Web Computing

Routr

Authors:	*ID Numbers:*
Elliott Moore	n9433236
Joshua Miles	n7176244

Unit Coordinator:
Associate Prof. Yue Xu

May 30, 2016

Generated using the LaTeX 2_ε typesetting system.

Table of Contents

Summary

Home Screen

User

.1 Create New User

.2 Logging in Exisiting user

.3 Logging out user

.4 Unregestered User unable to log in

.5 Review from user

Database

f. Searching for an item that exists in the database;

.1 Item within the database

g. Searching for an item that does not exist in the database;

.2 Item not within the database

.3 Attempted SQL Injection

h. Accessing an individual item page;

Pages

.1 Individual Item page

i. Attempting to use a cross site scripting attack but not being successful;

Multiple resolutions

Part A | **Brute Force Psudocode**

```
ALGORITHM BruteForceMedian(A[0..n-1])
k <- |n 2|
for i in 0 to n-1 do
numsmaller <- 0
numequal <- 0
for j in 0 to n-1 do
if A[j] < A[i] then
numsmaller <- numsmaller + 1
else
if A[j] = A[i] then
numequal <- numequal + 1
if numsmaller < k and k <= (numsmaller + numequal) then
return A[i]
```

Figure 1

Part B | **Median Psudocode**

```
ALGORITHM Median(A[0..n-1])
if n = 1 then
return A[0]
else
Select(A, 0, |_n/2_|, n-1)
```

Figure 2

Part C | **Select Psudocode**

```
ALGORITHM Select(A[0..n-1])
pos <- Partition(A,l,h)
if pos = m then
return A[pos]
if pos > m then
return Select(A, l, m, pos - 1)
if pos < m then
return Select(A, pos + 1, m, h)
```

Figure 3

Part D | **Partition Psudocode**

```
ALGORITHM Partition(A[0..n-1])
for j in l + 1 to h do
if A[j] < pivotal then
pivotloc <- pivotloc + 1
swap(A[pivotloc], A[j])
swap(A[l], A[pivotloc])
return pivotloc
```

Figure 4: Partitions array slice A[l..h] by moving element A[l] to the position it would have if the array slice was sorted, and by moving all values in the slice smaller than A[l] to earlier positions, and all values larger than or equal to A[l] to later positions. Returns the index at which the ‘pivot’ element formerly at location A[l] is placed.

Part E | **Brute Force Implementation - Basic Operations**

```
1  int BruteForceMedianOps(vector<int> unsortedArray) {
2  int lengthOfArray = (int) unsortedArray.size();
3  int k = lengthOfArray / 2;
4  basicOperations = 0;
5  for (int i = 0; i < lengthOfArray - 1; ++i) {
6  int numSmaller = 0;
7  int numEqual = 0;
8  for (int j = 0; j < lengthOfArray - 1; ++j) {
9  basicOperations++;
10 if (unsortedArray[j] < unsortedArray[i]) {
11 numSmaller++;
12 } else {
13 if (unsortedArray[j] == unsortedArray[i]) {
14 numEqual++;
15 }
16 }
17 }
18 if (numSmaller < k && k <= (numSmaller + numEqual)) {
19 return unsortedArray[i];
20 }
21 }
22 return -1;
23 }
```

Figure 5: The Brute Force Median finding algorithm which includes finding the amount of basic operations initialised on line 4 and incremented on line 9.

Part F | **Brute Force Implementation - Timed**

```
 1  int BruteForceMedianTime(vector<int> unsortedArray
      ) {
 2  int lengthOfArray = (int) unsortedArray.size();
 3  double k = ceil(lengthOfArray / 2.0);
 4  auto start = Clock::now();
 5  for (int i = 0; i < lengthOfArray - 1; ++i) {
 6  int numSmaller = 0;
 7  int numEqual = 0;
 8  for (int j = 0; j < lengthOfArray - 1; ++j) {
 9  if (unsortedArray[j] < unsortedArray[i]) {
10  numSmaller++;
11  } else {
12  if (unsortedArray[j] == unsortedArray[i]) {
13  numEqual++;
14  }
15  }
16  }
17  if (numSmaller < k && k <= (numSmaller + numEqual)
      ) {
18  auto end = Clock::now();
19  runtime = end - start;
20  return unsortedArray[i];
21  }
22  }
23  return -1;
24  }
```

Figure 6: This function uses the C++ languages standard library clock function on line 4 to begin timing a majority of the Brute Force Algorithm. The clock is than stopped at line 18 after the median has been located, fully encaptulating time to perform the algorithm from any other process.

Part G | Partition Implemenation - Basic Operations

```
int partition_count(vector<int> &array, int l, int
    h){
int temp;
int pivot_value = array[l];
int pivot_location = l;
for (int j = l + 1; j < h; j++){
basicOperations++;
if (array[j] < pivot_value){
pivot_location++;
temp = array[j];
array[j] = array[pivot_location];
array[pivot_location] = temp;
}
}
temp = array[pivot_location];
array[pivot_location] = array[l];
array[l] = temp;
return pivot_location;
}
```

Figure 7

Part H | **Select Implementation - Basic Operations**

```
int select_count(vector<int> &array, int l, int m,
    int h){
int pos = partition_count(array, l, h);
if (pos == m){
return array[pos];
}
if (pos > m){
return select_count(array, l, m, pos - 1);
}
return select_count(array, pos + 1, m, h);
}
```

Figure 8

Part I | **Median Implementation - Basic Operations**

```
int selection_median_count(vector<int> &array){
if (array.size() == 1){
return array[0];
}
return select_count(array, 0, (int)array.size()/2,
    (int)array.size() - 1);
}
```

Figure 9

Part J | **Brute Force Average Efficiency - Basic Operation**

We can show that the count of the operations in the average case will be:

$$\left\lceil \frac{n^2}{4} \right\rceil$$

where n is the size of the array. For each element less than the median, a full n loops must complete. This is looping through n elements. In the average case, the median element must exist in the middle of the array, and the algorithm stops once it has found the median. The middle of the array is at $\frac{n}{2}$ if there are n elements looped through, until $\frac{n}{4}$ loops have completed, that is $\frac{n}{2}$ repeats n times, or $\frac{n}{2}n * n = \frac{n^2}{4}$. Unfortunatley, this does not hold completely true. In cases where n is odd the count of operations is:

$$\left\lceil \frac{n^2}{4} \right\rceil$$