**NAME : BIPLAB DAS ROLL NO : 222CS2268**

**Q.1. Coin Tossing**

Through the simulation, show that probability of getting HEAD by tossing a fair coin is about 0.5. Write your observation from the simulation run.

**Q.2. Performance analysis of Bubble Sort**

Write the program to implement two different versions of bubble sort( BUBBLE SORT that terminates if the array is sorted before n-1th Pass. Vs. BUBBLE SORT that always completes the n-1th Pass) for randomized data sequence.

**Q.3. Variants of QUICK SORT**

Compare the performance of Quick Sort and Insertion Sort for instance characteristic n=10, ..., 1000. Find out the cross-over point where Insertion Sort shows the better performance over Quick Sort. Modify your sorting algorithm to stop partitioning the list in Quick Sort when the size of the (sub) list is less than or equal to 12 (say) and sort the remaining sub list using Insertion Sort. Your counter will now have to count compares in both the partition function and every iteration of Insertion Sort. Again, run the experiment for 50 iterations and record the same set of statistics. Compare your results for the two different sorting techniques and comment upon your results.

Q.1.

%Coin Toss:

x=zeros(1,1000);

y=zeros(1,1000);

td=1;

for n=1:10:10000

a=round(rand(1,n));

p=0;

x(td)=n;

for i=1:n

if a(i)==1

p=p+1;

end

end

p=p/n;

y(td)=p;

td=td+1;

end

% Coin toss simualtion

plot(x,y)

title('Coin Toss simulation run')

xlabel('Coin Toss number')

ylabel('Probability of HEAD')

grid on

%Simulation:



OBSERVATION :

• From the graph obtained by performing coin toss simulation run with 10,000 number of trials, it can be inferred that probability of getting head is nearly equivalent to 0.5

• It is observed that the graph approaches 0.5 with increase in the number of trials. This indicates heads and tails are equally probable in tossing a fair coin.

Q.2.

% bs1 function stands for bubble sort

function [comp] = bs1(a,n)

comp=0;

for i=1:n

for j=1:n-i

comp=comp+1;

if a(j)>a(j+1)

td=a(j);

a(j)=a(j+1);

a(j+1)=td;

end

end

end

end

% bs2 function stands for modified bubble sort

function [comp] = bs2(a,n)

comp=0;

for i=1:n

c=0;

for j=1:n-i

comp=comp+1;

if a(j)>a(j+1)

c= c+1

td=a(j);

a(j)=a(j+1);

a(j+1)=td;

end

end

if c==0

break

end

end

end

% Performance Analysis for Bubble-Sort and ModifiedBubbleSort

clear all

xb = zeros(1,10);

ybs1 = zeros(1,10);

ybs2 = zeros(1,10);

comp=0;

td=0;

k=1;

for n=10:10:100

xb(k)=n

a=round(rand(1,n)\*100)

%................Importing Bubble sort functions........

ybs1(k)=bs1(a,n) %Normal Bubble sort %Time Complexcity O(n\*n)

ybs2(k)=bs2(a,n) %Modified Bubble sort %Time Complexcity O(n)

k=k+1;

end

%...............Plotting graph......................

plot(xb,ybs1,xb,ybs2)

legend('Bubble Sort O(n\*n)','Modified Bubble Sort O(n)')

title('Performance Analysis of Bubble Sort')

xlabel('Number of elements in Array')

ylabel('Number of comparisons')

grid on



Q.3.

………VARIANTS OF QUICK SORT………………………………………………………..

INSERTION SORT AND VARIANTS OF QUICK SORT:

1) NORMAL QUICK SORT

2) RANDOMIZED QUICK SORT (Here the pivot element in chosen randomly each time)

3) QUICK-INSERTION SORT (As per the conditions provided in the question) COMPARISON AND PERFORMANCE ANALYSIS OF INSERTION SORT, QUICK SORT AND QUICKINSERTION SORT.



% quickSort

function[a, cnt] = quickSort(a, l, r, cnt)

if(l<r)

[a,mid,cnt] = partition(a,l,r);

cnt = cnt + 1;

c=0;

[a,c1] = quickSort(a, l, mid-1,c);

[a,c2] = quickSort(a, mid+1,r,c);

cnt= cnt + c1+c2;

end

end

%The partition function

function[a, mid, count] = partition(a, l, r)

pivot = a(r);

i=l-1;

count=0;

for j=l:l-1

count=count+1;

if a(j)<pivot

i=i+1;

temp=a(i);

a(i)=a(j);

a(j)=temp;

end

end

temp = a(i+1);

a(i+1)=a(r);

a(r)=temp;

mid=i+1;

%Randomized-QuickSort function

function[a,count] = QuickSort(a, l, r,count)

if(l<r)

[a,mid,count] = partitionRandom(a,l,r);

count =count + 1;

c=0;

[a,c1] = QuickSort(a, l, mid-1,c);

[a,c2] = QuickSort(a, mid+1,r,c);

count=count + c1+c2;

end

end

%The partition function for Randomized quick sort

function[a, mid, count] = partitionRandom(a, l, r)

pos = l + round(rand(l)\*(r-1));

temp = a(pos);

a(pos) = a(l);

a(l)= temp;

pivot = a(l);

pi=1;

count=0;

for j=l+1:r

count=count+1;

if a(j)<=pivot

temp=a(j);

a(j)=a(pi+1);

a(pi+1)=temp;

pi=pi+1;

end

end

temp = a(pi);

a(pi)=pivot;

a(l)=temp;

end

% SOURCE CODE FOR QUICK-INSERTION SORT

function [comp] = quickInsertion(a,n)

if (n<=12)

comp = insertionSort(a,n);

else

[a,z]=quickSort(a,1,n,0);

comp = z;

end

end

OBSERVATION : • From the previous question (Q-3) we came to know the performance of quick sort and insertion sort on different type of data inputs. There are conditions when insertion sort is efficient too.



• To form an enhanced sorting algorithm, we test insertion sort and quick sort to find an intersection point and hence form and new enhanced sort. The cross over point achieved is 12.(as per the question)

• We know insertion sort is efficient when there is repeated or almost sorted data, it is observed that the new enhanced sorting algorithm i.e Quick-Insertion sort performs better than quick sort when the number of elements are increased.

--------------------------------THANK YOU--------------------------------