**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.

Code:

**assignment\_1.m**

noelt=zeros(1,200);

p=zeros(1,200);

k=1;

for i = 10:500:100000

tosses = randi([0,1],[i,1]);

noelt(k) = i;

a = 0;

b = 0;

for z =1:10

for j = 1:i

if tosses(j) == 0

a = a + 1;

else

b =b + 1;

end

end

end

p(k) = a/(i\*10);

k = k+ 1;

end

plot(noelt,p);

xlabel("#Tosses");

ylabel("Probability of Geting head");

grid on;



**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. Average case analysis for Sorting Algorithms**

For each of the data formats: random, reverse ordered, and nearly sorted, run your program say **SORTTEST** for all combinations of sorting algorithms and data sizes and complete each of the following tables. When you have completed the tables, analyze your data and determine the asymptotic behavior of each of the sorting algorithms for each of the data types *(i)* ***Random data, (ii) Reverse Ordered Data, (iii) Almost Sorted Data*  and (iv) *Highly Repetitive Data*** *.* select the suitable no of elements for the analysis that supports your program. Select at least five internal sorting algorithm to present your analysis









**Q.4. Variants of QUICK SORT**

Compare the performance of **variants of quick sort** algorithm for instance characteristic n=10, ..., 1000. Use the finding from Q3, [cross-over point where insertion sort shows the better performance over quick sort] Modify your sorting algorithm in the previous problem to stop partitioning the list in QUICKSORT when the size of the (sub)list is less than or equal to 12 and sort the remaining sublist using INSERTIONSORT. Your counter will now have to count compares in both the partition function and in each iteration of INSERTIONSORT. 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.

Code:

**assignment\_4.m**

noelt=zeros(1,50);

nocp\_1=zeros(1,50);

nocp\_2 = zeros(1,50);

k=1;

for n=10:20:1000

noelt(k)=n;

av\_cp\_1 = 0;

av\_cp\_2 = 0;

for i=1:10

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

av\_cp\_1 = av\_cp\_1 + quicksort(a)

av\_cp\_2 = av\_cp\_2 + quicksort\_1(a)

end

nocp\_1(k) = av\_cp\_1 / 10;

nocp\_2(k) = av\_cp\_2 / 10;

k=k+1

end

**Quick Sort Normal**

function comp = quicksort(x)

comp = 0

n = length(x);

[x comp] = quicksorti(x,1,n,comp);

end

function [x comp] = quicksorti(x,ll,uu,comp)

comp = comp + 1

if (ll < uu )

[x mm comp] = partition(x,ll,uu,comp);

[x comp] = quicksorti(x,ll,mm - 1,comp);

[x comp] = quicksorti(x,mm + 1,uu,comp);

end

end

function [x mm comp] = partition(x,ll,uu,comp)

pp = 1

x = swap(x,ll,pp);

mm = ll;

for j = (ll + 1):uu

comp = comp + 1

if (x(j) < x(ll))

mm = mm + 1;

x = swap(x,mm,j);

end

end

x = swap(x,ll,mm);

end

function x = swap(x,i,j)

val = x(i);

x(i) = x(j);

x(j) = val;

end

**Quick Sort with Insertion Sort**

function comp = quicksort\_1(x)

comp = 0

kk = 12; % Insertion sort threshold, kk >= 1

n = length(x);

[x comp] = quicksorti(x,1,n,kk,comp);

end

function [x comp] = quicksorti(x,ll,uu,kk,comp)

[x mm comp] = partition(x,ll,uu,comp);

comp = comp + 1;

if ((mm - ll) <= kk)

[x comp] = insertionsorti(x,ll,mm - 1,comp);

else

[x comp] = quicksorti(x,ll,mm - 1,kk,comp);

end

comp = comp + 1;

if ((uu - mm) <= kk)

[x comp] = insertionsorti(x,mm + 1,uu,comp);

else

[x comp] = quicksorti(x,mm + 1,uu,kk,comp);

end

end

function [x mm comp] = partition(x,ll,uu,comp)

pp = 1;

x = swap(x,ll,pp);

mm = ll;

for j = (ll + 1):uu

comp = comp + 1;

if (x(j) < x(ll))

mm = mm + 1;

x = swap(x,mm,j);

end

end

x = swap(x,ll,mm);

end

function [x comp] = insertionsorti(x,ll,uu,comp)

for j = (ll + 1):uu

pivot = x(j);

i = j;

comp = comp +1

while ((i > ll) && (x(i - 1) > pivot))

comp = comp + 1;

x(i) = x(i - 1);

i = i - 1;

end

x(i) = pivot;

end

end

function x = swap(x,i,j)

val = x(i);

x(i) = x(j);

x(j) = val;

end



**Q. 5 Variants of insertion SORT**

The **two way insertion sort** is a modification of your simple insertion sort. Suppose you are sorting the array x. A seperate output array of size 2n+ 1 is set aside. Initially x[0] is placed into the middle element of the array n. Continue inserting elements until you need to insert between a pair of elements in the array. As before you need to make room for the new element by shifting elements. Unlike before, you can choose to shift all smaller elements one step to the left or all larger elements one step to the right since there is additional room on both sides of the array. The choice of which shift to perform depends on which would require shifting the smallest amount of elements. Compare the performance of two way insertion sort with quick sort.

Code:

**assignment\_5.m**

noelt=zeros(1,10);

nocp=zeros(1,10);

nocp\_1=zeros(1,10);

k=1;

for n=10:10:100

noelt(k)=n;

av\_cp\_1 = 0;

av\_cp\_2 = 0;

for i=1:10

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

av\_cp\_1 = av\_cp\_1 + twowayinsertionsort(a)

av\_cp\_2 = av\_cp\_2 + quicksort(a)

end

nocp(k) = av\_cp\_1 / 10;

nocp\_1(k) = av\_cp\_2 / 10;

k=k+1

end

plot(noelt,nocp,noelt,nocp\_1)

xlabel('#elements')

ylabel('#comparisons')

legend('Two way insertion sort','quicksort')

grid on

**Two Way Insertion Sort**

function comp = twowayinsertionsort(x)

comp = 0;

n = length(x);

a = zeros(1,2\*n+1);

for i=1:n

a(n+i) = x(i);

end

l = n+1;

h = n+1;

disp(a);

for i=n+2:2\*n

comp = comp +1;

if a(h) <= a(i)

h = h + 1;

a(h) = a(i);

comp = comp +1;

if h ~= i

a(i) = 0;

end

elseif a(l) >= a(i)

comp = comp +1;

l = l - 1;

a(l) = a(i);

a(i) = 0;

else

j = h - 1

while a(i) < a(j)

comp = comp +1;

j = j-1

end

lc = j-l +1;

rc = h-j;

comp = comp +1;

if rc < lc

h = h+1

for k=h:-1:j+1

a(k) = a(k-1);

end

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

a(i) = 0;

end

end

end

end



**Q.6 QUICK SELECT**

Use the **QUICK SELECT** algorithm to find **3rd largest element** in an array of n integers. Analyze the performance of **QUICK SELECT** algorithm for the different instance of size 50 to 500 element. Record your observation with the *number of comparison made* vs. *instance*.

Code:

**assignment\_6.m**

noelt=zeros(1,10);

nocp=zeros(1,10);

k=1;

for n=50:10:500

noelt(k)=n

cmp = 0

for i=1:10

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

cmp= cmp + quickselect(a,3)

end

nocp(k)=cmp/10

k=k+1

end

plot(noelt,nocp)

xlabel('#elements')

ylabel('#comparisons')

legend('Quick Select')

grid on

**Quick Select**

function comp = quickselect(A,k)

comp = 0

[elem comp] = iquickselect(A,k,comp)

end

function [elem comp] = iquickselect(A,k,comp)

n = length(A)

r = randi(n)

pv = A(r)

A1 = []

A2 = []

for i = 1:n

comp = comp +2

if A(i) < pv

A1 = [A1 A(i)]

elseif A(i) > pv

A2 = [A2 A(i)]

else

end

end

comp = comp + 2

if k <= length(A1)

[elem comp] = iquickselect(A1,k,comp)

elseif k > length(A) - length(A2)

[elem comp] = iquickselect(A2, k - (length(A) - length(A2)),comp)

else

elem = pv

end

end



**Q.7 BINARY INSERTION SORT**

**Binary Insertion sort** is a variant of Insertion sorting in which proper location to insert the selected element is found using the binary search. Compare the performance of Binary insertion sort with Quick sort and insertion sort. Run the experiment for at least 10 iterations for an instance and compare your results and comment on your simulation results.

CODE:

**assignment\_7.m**

noelt=zeros(1,10);

nocp=zeros(1,10);

nocp\_1=zeros(1,10);

nocp\_2=zeros(1,10);

k=1;

for n=10:10:100

noelt(k)=n

cmp= 0

cmp\_1 = 0

cmp\_2 = 0

for i=1:10

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

cmp= cmp + binaryinsertionsort(a)

cmp\_1 = cmp\_1 + quicksort(a)

cmp\_2 = cmp\_2 + insertionSort(a)

end

nocp(k)=cmp/10

nocp\_1(k) = cmp\_1/10

nocp\_2(k) = cmp\_2/10

k=k+1

end

plot(noelt,nocp,noelt,nocp\_1,noelt,nocp\_2)

xlabel('#elements')

ylabel('#comparisons')

legend('binaryInsertionSort',"quickSort","insertionSort")

**Binary insertion Sort**

function comp = binaryinsertionsort(list)

comp = 0

for i = (2:numel(list))

value = list(i);

j = i - 1;

[index comp] = binarysearch(list,j,value,comp)

while j >=index

list(j+1) = list(j);

j = j-1;

%comp= comp + 1

end

list(j+1) = value;

end

end

function [index comp] = binarysearch(A,ll,num, comp)

left = 1;

right = ll;

flag = 0;

while left <= right

comp = comp + 1

mid = ceil((left + right) / 2);

comp = comp + 2

if A(mid) == num

index = mid;

flag = 1;

break;

else if A(mid) > num

right = mid - 1;

else

left = mid + 1;

end

end

end

comp = comp+1

if flag == 0;

index = left;

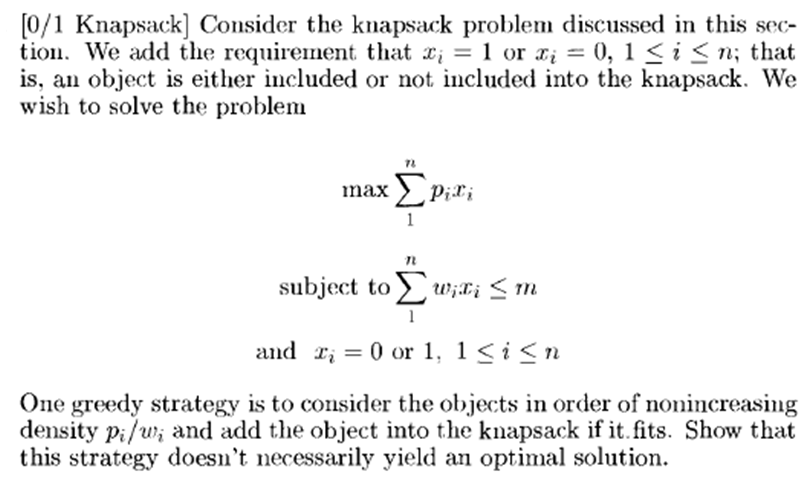
end

end



**You are free to select programming language/ environment to complete the following assignments**

**Q.8 0/1 KNAPSACK PROBLEM**



You should write two different algorithms (using subset sum paradigm and ordering paradigm) and compare their performance by varying knapsack capacity from 1 to maximum size (Twice the sum of all weights of individual items).

**Code:**

**assigment\_8.m**

k=1;

P = [60, 100, 120]

W = [10, 20, 30]

max\_size = 2\*sum(W)

nocp\_1=zeros(1,max\_size);

nocp\_2=zeros(1,max\_size);

for n=1:max\_size

nocp\_1(k) = oderingknapsack(W,P,n)

nocp\_2(k) = knapsacksubset(W,P,n)

k=k+1

end

plot([1:max\_size],nocp\_1,[1:max\_size],nocp\_2)

xlabel('#capacity')

ylabel('#profit')

legend("Odering","Subset")

function bagv = oderingknapsack(W,P,C)

ps = P./W

[Y I] = sort(ps,'descend')

bag = []

bagv = 0

bagw = 0

n = length(P)

i = 1

if min(W) > C

return

end

while bagw < C && i < n

bagv = bagv + P(I(i))

bagw = bagw + W(I(i))

bag = [bag I(i)]

i = i + 1

end

end

function prof = knapsacksubset(W,P,C)

prof = knapsacksubetr(W,P,C)

end

function data = knapsacksubetr(W,P,C)

n = length(P)

if n == 0 || C == 0

data = 0

return

end

if W(n) > C

data = knapsacksubetr(W(1:n-1),P(1:n-1),C)

else

data = max(P(n) +knapsacksubetr(W(1:n-1),P(1:n-1),C-W(n)),knapsacksubetr(W(1:n-1),P(1:n-1),C))

end

end

