ASSIGNMENT - I

CS 6381:
ADVANCED PROGRAMMING LABORATORY

SHILPA S
(220CS1256)

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.

1.1 Algorithm:

Algorithm 1 Algorithm to simulate that probability of getting HEAD by tossing a fair coin is about 0.5.

```
1.Initialize noe and p to 1 by 1000 zero Matrix and k to 1.
2. for each i value from 10 to 10000 with 10 step value
do the following
  2.1 assign noe(k) to i
  2.2 generate a matrix of dimension 1 by i having random
  2 digit values.
  2.3 initialize head to 0;
  2.4 for each j value from 1 to i do the following
     2.4.1 check trails(j) modulus with 2
            2.4.1.1 if it is equal to 0, increment head by 1
  2.5 assign p(k) to head/i
  2.6 increment k by 1
3 plot noe and p
4 give proper title , xlabel and ylabel
```

1.2 Program code:

```
noe=zeros(1,1000);
p=zeros(1,1000);
k=1;
for i=10:10:10000
    noe(k)= i;
    trails = round(rand(1,i)*100);
    head=0;
    for j=1:i
        if modulo(trails(j) , 2) == 0
            head=head+1;
        end
    end
    p(k)=head/i;
    k=k+1;
end
plot(noe,p,'k');
```

```
xgrid(2);
title("Probability of getting HEAD by tossing a fair coin.");
xlabel("Number of trails");
ylabel("Probability of getting HEAD");
```

1.3 Interpretation:

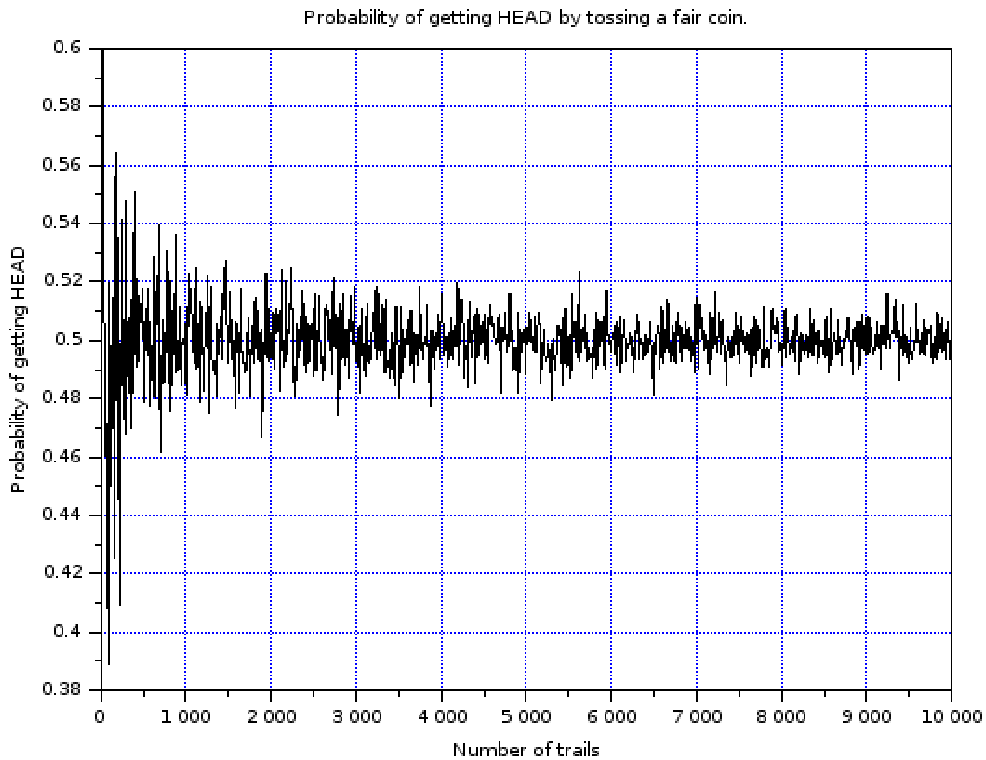


Figure 1.1

1.4 Comments:

Figure 1.1, shows that the probability of the heads oscillates between a range. As the number of tosses increases the range converges to 0.5.

The total number of samples taken is 10000. We can see from the graph that during the initial 200 samples, the probability varies beyond (0.4, 0.6) range. This is because the probability only tells the likelihood of number of heads that may appear in n coin tosses and not the actual number of heads.

The probability formula by frequentist approach is:

$$\text{P(X= HEADS)} = T/N$$

Where,
T = Number of times heads appears in N coin tosses .
N =N coin tosses

If N = 200 then only a deviation of 20 coin-tosses will shift the probability by 0.1. The impact of such deviations is reduced as we increase the number of tosses.

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-1 th Pass. Vs. BUBBLE SORT that always completes the n-1 th Pass) for randomized data sequence.

2.1 Algorithm::

Algorithm 2 Algorithm to implement two different versions of bubble sort

```
1.initialize noe, noc and nc to 1 by 5 zero matrix
2.initialize cmp,cp and t to 0
3.initialize k to 1
4 for each n value from 10 to 50 with a step value 10
   4.1 assign noe(k) to n
   4.2 generate a 1 by n matrix having random 2 digit numbers
   and assign it to a
   4.3 copy matrix a to matrix b

   //Regular BubbleSort

   4.4 for each value of i from 1 to n
       4.4.1 for each value of j from 1 to n-i
           4.4.1.1 increment cmp by 1
           4.4.1.2 compare a(j) with a(j+1),if it is
           greater than a(j+1), swap the values.
  4.5 assign noc(k) to cmp and cmp to 0

  //modified bubble sort

  4.6 for each value of i from 1 to n
       4.6.1 set flag to 0
       4.6.2 for each value of j from 1 to n-1
            4.4.2.1 compare a(j) with a(j+1),if it is greater
            than a(j+1),
                4.4.2.1.1 swap the values.
                4.4.2.1.2 imcrement cp by 1
                4.4.2.1.3 set flag to 1
      4.6.3 if flag is equal to 0, brea
  4.5 assigkn nc(k) to cp and cp to 0
 5 plot noe and noc
 6 plot noe and nc
 7 use proper xlabel,ylabel , title and legend
```

2.2 Program Code:

```
noe=zeros(1,5);
noc=zeros(1,5);
nc=zeros(1,5);
cmp=0;
cp=0;
t=0;
k=1;
for n=10:10:50
    noe(k)=n;
    for z=1:10
    a=round(rand(1,n)*100);
   //copy matrix a to b
    b=zeros(1,n);
      for s=1:n
        b(s)=a(s);
      end
  //regular bubbleSort
      for i=1:n
        for j=1:n-i
            cmp=cmp+1;
            if a(j)>a(j+1)
                t=a(j);
                a(j)=a(j+1);
                a(j+1)=t;
            end
        end
      end
  noc(k)=cmp;
  cmp=0;
 //modified bubbleSort
 for i=1:n
     flag=0;
     for j=1:n-1
          if b(j) >b(j+1)
               cp=cp+1;
                t=b(j);
                b(j)=b(j+1);
                b(j+1)=t;
                flag=1;
          end
     end
     if flag == 0
         break;
      end
```

```
    end
    nc(k)=cp;
    cp=0;
  end
noc(k)=noc(k)/10;
nc(k)=nc(k)/10;
  k=k+1;
end
plot(noe,noc,'+-+ k');
plot(noe,nc,'*-* r');
xlabel('input element size');
ylabel('no of comparison');
title('Bubble sort performance analysis');
xgrid(2)
legend('BubbleSort','ModifiedBubbleSort',2);
```

2.3 Conclusion:

From Figure 2.1, we can conclude that both the variants of bubble sort takes $\mathbf{O}(n^2)$ time. The only difference is that the modified bubble sort takes less time compared to regular it is because of early exit from the sorting algorithm when array is found to be sorted.

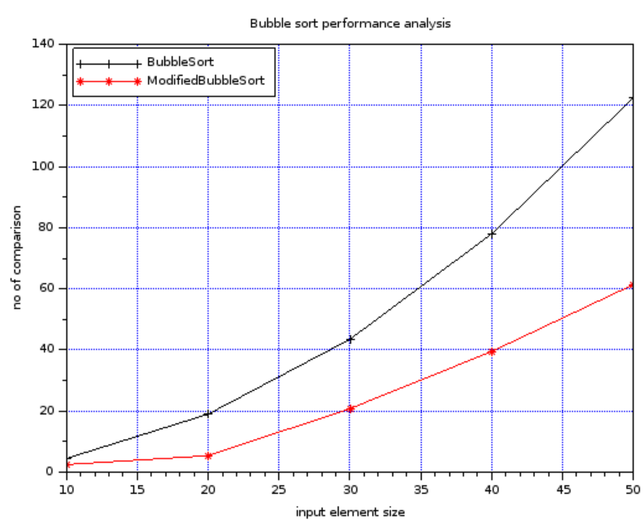


Figure 2.1

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.

3.1 Algorithm:

Algorithm 3 Insertion Sort

```
1. Sorttest_insertion(a)
    1.1 Initialize noc to 0 and find length of a and assign it
    to n
    1.2. For each i value from 2 to n
        1.2.1  Assign key as a(i) and j as i-1
        1.2.2  While j is greater than or equal to 1
               and  a(j) is greater than key, repeat
               the following
                1.2.2.1 Increment noc.
                1.2.2.2 Assign a(j) to a(j+1)
                1.2.2.3 Decrement j
        1.2.3 Assign key to a(j+1)
    1.3 return noc and a.
```

Algorithm 4 Selection Sort

```
1. Sorttest_selection(a)
    1.1 Initialize noc to 0 and find length of a and assign
    it to n
    1.2. For each i value from 1 to n-1
        1.2.1  Initialize minIndex to i
        1.2.2  For each j value from i+1 to n
               1.2.2.1 Check whether a(j) is less
               than a(minIndex)
                       1.2.2.1.1 If it is true, then
                       increment noc
and assign minIndex
                       to j
        1.2.3 Swap  a(minIndex) and a(i)
    1.3 return noc and a.
```

Algorithm 5 Bubble Sort

```
1. Sorttest_bubble(a)
    1.1 Initialize noc to 0 and find length of a and assign
    it to n
    1.2. For each i value from 1 to n
        1.2.1  Set flag to 0.
        1.2.2  For each j value from 1 to n-1
               1.2.2.1 increment noc
               1.2.2.1 Check whether a(j) is greater
                       than a(j+1), If it is true, then
                       swap a(j) and a(j+1) and  set
                       flag to 1.
        1.2.3 Check if flag is equal to 0, if yes
              then break.
    1.3 return noc and a.
```

Algorithm 6 Quick Sort

```
/* low  —> Starting index,  high  —> Ending index */
1.quickSort(arr[], low, high)
  1.1 if (low < high) do the following
     1.1.1 call partition function with parameters arr,
           low and high and it returns partitioning
           index and element at that index is at right
           place.
     1.1.2 do quicksort recursively for left partition.
     1.1.3 do  quicksort recursively for right partition.
```

Algorithm 7 Quick -partition

```
1.partition (arr[], low, high)
  1.1 initialize pivot to arr[high] and i to
     (low - 1)  // Index of smaller element
  1.2 for j values from low to high-1 do the following
     1.2.1 if arr[j] <pivot then
          1.2.1.1 increment i
          1.2.1.2 swap arr[i] and arra[j]
     1.2.2 swap arr[i + 1] and arr[high]
  1.3 return (i + 1) as partitioning index
```

3.2 Program Code:

```
//Sorttest_bubble function
function [a,cp]= sorttest_bubble(a)
    cp=0;
    n=length(a);
   for i=1:n
    flag=0;
     for j=1:n-1
             cp=cp+1;
          if a(j) > a(j+1)
                t=a(j);
                a(j)=a(j+1);
                a(j+1)=t;
                flag=1;
          end
     end
     if flag == 0
         break;
      end
  end
endfunction

//Sorttest_insertion function
function [a,noc]= sorttest_insertion(a)
    noc=0;
    n=length(a);
  for i=2:n
    key = a(i);
    j = i-1;
    while (j>=1 && a(j) >key)
        noc=noc+1;
        //disp("h");
        a(j+1) = a(j);
        j=j-1;
     end
     a(j+1) = key;
   end
endfunction

//Sorttest_selection function

function [a,noc]= sorttest_selection(a)
    noc=0;
    n=length(a);
  for i=1:n-1
```

```
    min_idx =i;
    for j=i+1 :n
          if a(j) < a(min_idx)
              noc=noc+1;
              min_idx =j;
          end
     end
     t=a(min_idx);
     a(min_idx)=a(i);
     a(i)=t;
end
endfunction
//partition function of quick sort

function[k,cmp,a]= partition(a,low,high,cmp)
    pivot = a(high);
    i = low-1;
    for j=low:high-1
        if a(j) <=  pivot then
            cmp=cmp+1;
            i=i+1;
            t=a(i);
            a(i)=a(j);
            a(j)=t;
        end
    end
    p= a(i+1);
    a(i+1) = a(high);
    a(high) = p;
    k=i+1;
endfunction

//quick sort function
function[cmp,a] = sorttest_quick(a,low,high,cmp)
    if low < high then
       [pi,cmp,a]=  partition(a,low,high,cmp);
        [cmp,a]=sorttest_quick(a,low,pi-1,cmp);
       [cmp,a]=  sorttest_quick(a,pi+1,high,cmp);
    end
endfunction

//main program
k=1;
noe = zeros(1,5);
noc_r=zeros(1,5);
noc_f=zeros(1,5);
```

```
noc_a=zeros(1,5);
noc_d=zeros(1,5);
noc_s = zeros(1,5);

nocs_r=zeros(1,5);
nocs_f=zeros(1,5);
nocs_a=zeros(1,5);
nocs_d=zeros(1,5);
nocs_s=zeros(1,5);

nocb_r=zeros(1,5);
nocb_f=zeros(1,5);
nocb_a=zeros(1,5);
nocb_d=zeros(1,5);
nocb_s=zeros(1,5);

//random number
for n=10:10:50
    noe(k)=n;
    for z=1:10
     h=round(rand(1,n)*100);
     b=zeros(1,n);
     c=zeros(1,n);

     [ arr_i, noc_r(k) ]=sorttest_insertion(h);
     [ arr_i, nocs_r(k) ]=sorttest_selection(h);
     [ arr_i, nocb_r(k) ]=sorttest_bubble(h);

     //sorted data
     [ arr_i, noc_s(k) ]=sorttest_insertion(arr_i);
     [ arr_i, nocs_s(k) ]=sorttest_selection(arr_i);
     [ arr_i, nocb_s(k) ]=sorttest_bubble(arr_i);

     //reversed ordered data
     b=flipdim(arr_i,2);
     [ arr_i, noc_f(k) ]=sorttest_insertion(b);
      [ arr_i, nocs_f(k) ]=sorttest_selection(b);
       [ arr_i, nocb_f(k) ]=sorttest_bubble(b);

    //almost sorted
    c=[1:n/2,round(rand(1,n/2)*100)];
    [ arr_i, noc_a(k) ]=sorttest_insertion(c);
    [ arr_i, nocs_a(k) ]=sorttest_selection(c);
     [ arr_i, nocb_a(k) ]=sorttest_bubble(c);

   //highly repeatative
```

```
d=grand(1,n,'uin',1,4);
  [ arr_i, noc_d(k) ]=sorttest_insertion(d);
  [ arr_i, nocs_d(k) ]=sorttest_selection(d);
  [ arr_i, nocb_d(k) ]=sorttest_bubble(d);
end
nocs_r(k)=nocs_r(k)/10;
nocs_a(k)=nocs_a(k)/10;
nocs_f(k)=nocs_f(k)/10;
nocs_d(k)=nocs_d(k)/10;
nocs_s(k)=nocs_s(k)/10;

noc_r(k)=noc_r(k)/10;
noc_a(k)=noc_a(k)/10;
noc_f(k)=noc_f(k)/10;
noc_d(k)=noc_d(k)/10;
noc_s(k)=noc_s(k)/10;

nocb_r(k)=nocb_r(k)/10;
nocb_a(k)=nocb_a(k)/10;
nocb_f(k)=nocb_f(k)/10;
nocb_d(k)=nocb_d(k)/10;
nocb_s(k)=nocb_s(k)/10;
k=k+1;
end

subplot(221)
plot(noe,noc_r,"b" , noe,noc_a,"y");
plot(noe,noc_f,"g",noe,noc_d,"r")
xlabel('input element size');
ylabel('no of comparison');
title('Insertion sort performance analysis');
xgrid(2)
legend('Random Data','Almost sorted', 'Reverse Order'
,'Highly repetitive', 2);

subplot(222)
plot(noe,nocs_r,"b" , noe,nocs_a,"y");
plot(noe,nocs_f,"g",noe,nocs_d,"r")
xlabel('input element size');
ylabel('no of comparison');
title('Selection sort performance analysis');
xgrid(2)
legend('Random Data','Almost sorted', 'Reverse Order'
,'Highly repetitive', 2);

subplot(2,2,3)
```

```
plot(noe,nocb_r,"b" , noe,nocb_a,"y");
plot(noe,nocb_f,"g",noe,nocb_d,"r")
plot(noe,nocb_s,"m");
xlabel('input element size');
ylabel('no of comparison');
title(' Bubble sort performance analysis');
xgrid(2)
legend('Random Data','Almost sorted',
'Reverse Order' ,'Highly repetitive','Sorted', 2);
```

3.3 Interpretation:

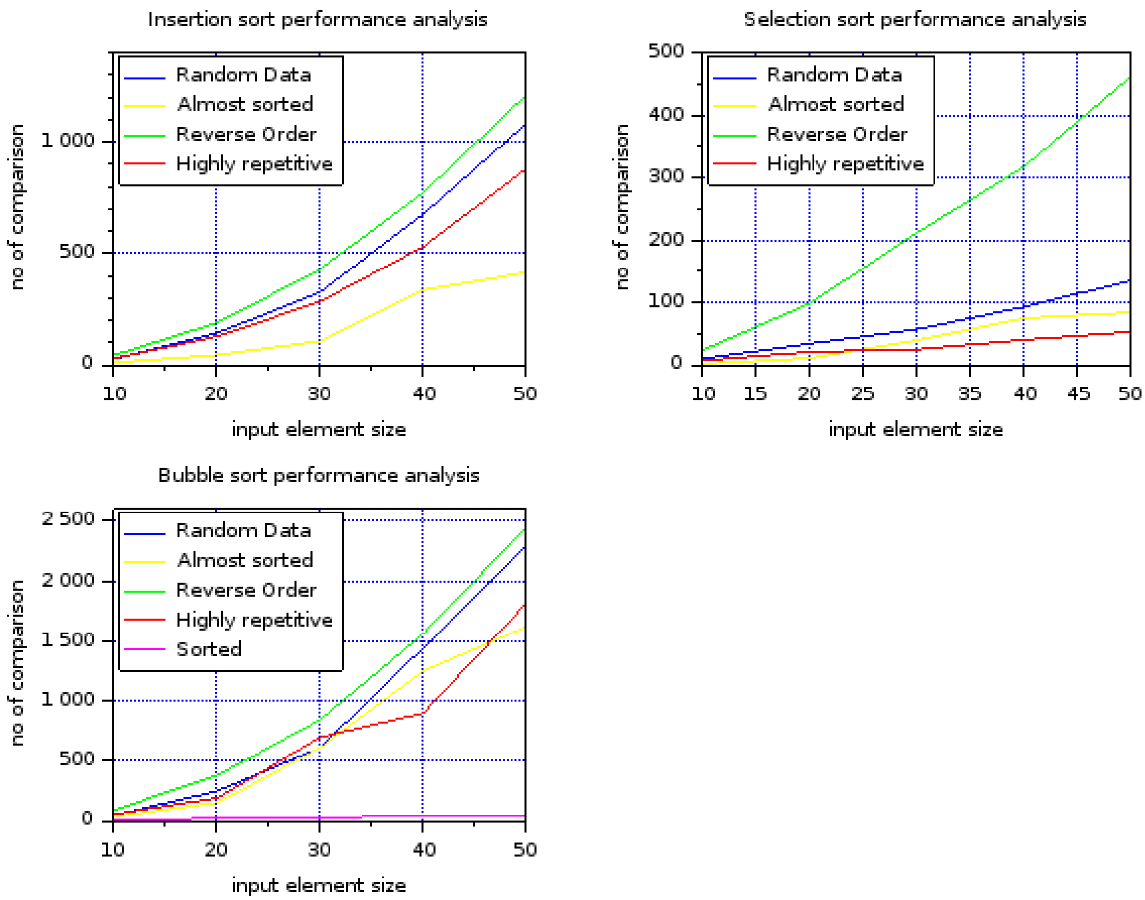


Figure 3.1: Different type of sorting algorithms

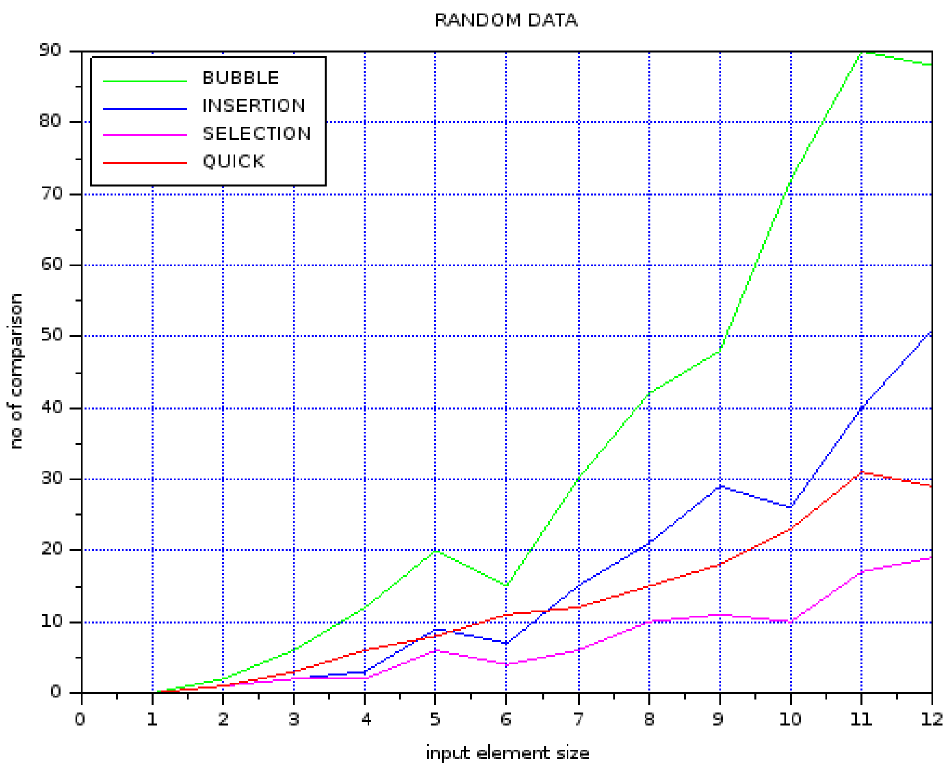
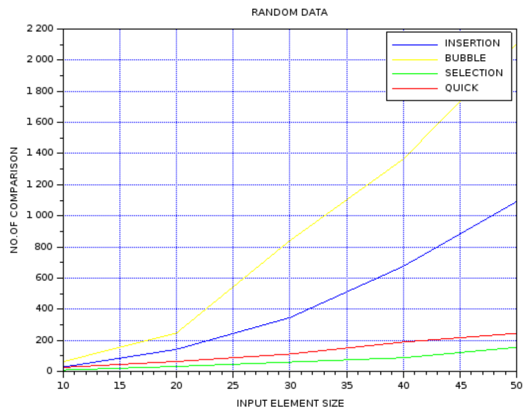
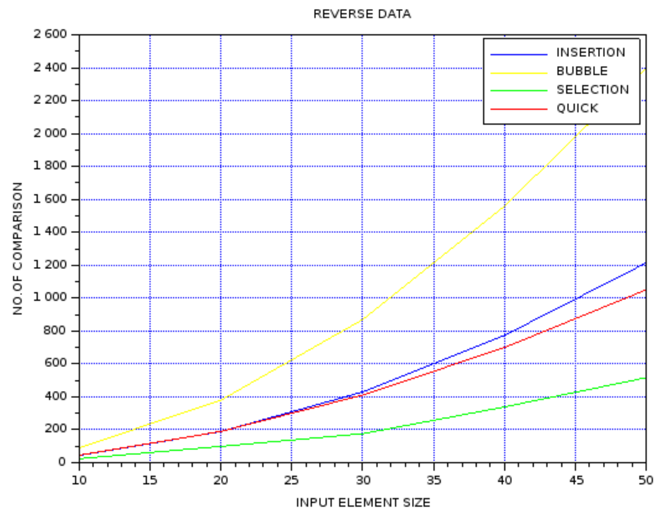


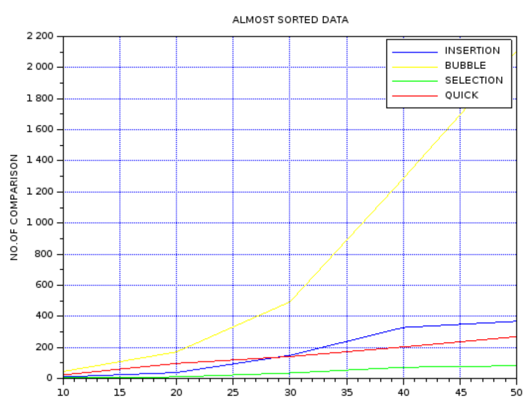
Figure 3.2: Crossover Point – quicksort vs insertion sort


RANDOM DATA
INSERTION
BUBBLE
SELECTION
QUICK
NO.OF COMPARISON
INPUT ELEMENT SIZE

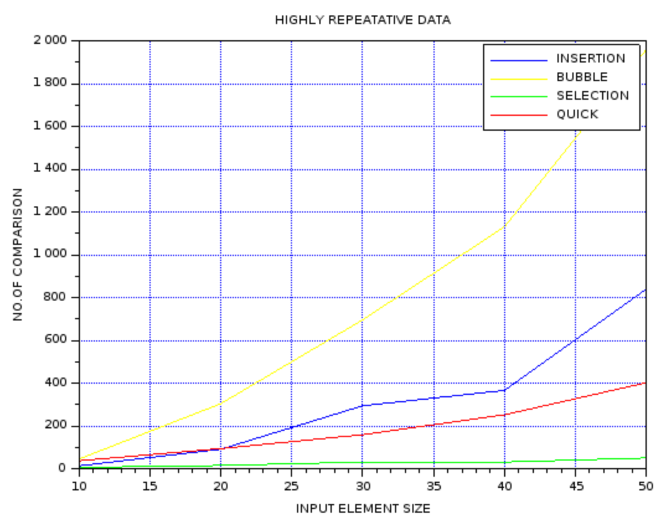
(a)


REVERSE DATA
INSERTION
BUBBLE
SELECTION
QUICK
NO.OF COMPARISON
INPUT ELEMENT SIZE

(b)


ALMOST SORTED DATA
INSERTION
BUBBLE
SELECTION
QUICK
NO.OF COMPARISON
INPUT ELEMENT SIZE

(a)


HIGHLY REPEATATIVE DATA
INSERTION
BUBBLE
SELECTION
QUICK
NO.OF COMPARISON
INPUT ELEMENT SIZE

(b)

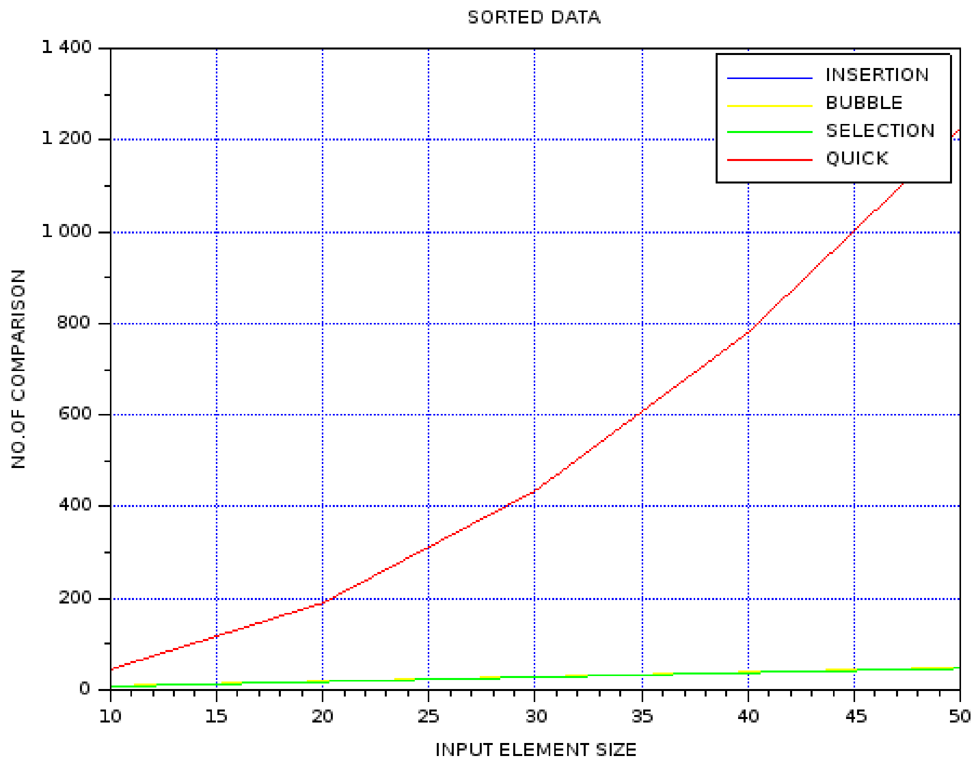


Figure 3.5

3.3.1 Comments:

- Figure 3.2 graph shows that crossover point between insertion sort and quick sort lies at the (4,7) and crossover point is 6.
- From figures 3.3a ,3.3b, 3.4a,3.4b we can conclude that selection sort performs best in all input cases and bubble sort perform worse in all input cases.
- But in case of sorted input, all sorting algorithm except quick performs well, it is shown in figure 3.5.
- From figure 3.1, we can conclude that for random data all sorting algorithms takes more time compared to all other type of inputs.

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 every 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.

4.1 Algorithm:

Algorithm 8 Hybrid Quick

```
1.quick(a,low,high,cmp)
   1.1 calculate length of a and assign it to len.
   1.2 if(high-low+1 > 12) then do the following
       1.2.1 if low < high then
             1.2.1.1 Call partition function
             with parameters a, low, high.
             1.2.1.2 call hybrid quick sort recursively
             over left partition.
            1.2.1.3 call hybrid quick sort recursively
             over right partition
```

4.2 Program Code:

```
\\hybrid quick function{i.e,when sublist is less
than or equal to 12 ,sort using insertion otherwise
normal quick}

function[cmp,a]=quick(a,low,high,cmp)
    //len = length(a);
    if(high-low+1 > 12) then
        if low < high then
            [pi,cmp,a]= hybrid_partition(a,low,high,cmp);
            [cmp,a]= quick(a,low,pi-1,cmp);
            [cmp,a]= quick(a,pi+1,high,cmp);
         end
    else
```

```
        [a,cmp]=sorttest_insertion(a,low,high,cmp);
    end
endfunction

\\hybrid partition function

function[k,cmp,a]= hybrid_partition(a,low,high,cmp)
    pivot = a(high);
    i = low-1;
    for j=low:high-1
        if a(j) <=  pivot then
            cmp=cmp+1;
            i=i+1;
            t=a(i);
            a(i)=a(j);
            a(j)=t;
        end
    end
    p= a(i+1);
    a(i+1) = a(high);
    a(high) = p;
    k=i+1;
endfunction

\\main program {for random - data}
k=1;
cp=zeros(1,991);
c=zeros(1,991);
noe=zeros(1,991);
for n=10:1000
    noe(k)=n;
    a=round(rand(1,n)*100);
    [cmp_h,ab] = quick(a,1,n,0);
    [cmp_r,ar] = sorttest_quick(a,1,n,0);
    cp(k) = cmp_r;
    c(k) = cmp_h;
    k=k+1;
end
plot(noe,cp,"m");
plot(noe,c);
xlabel("ARRAY SIZE");
ylabel("NUMBER OF COMPARISON");
title("FOR RANDOM DATA");
xgrid(2);
legend("NORMAL QUICK","HYBRID QUICK",2);
```

4.3 Intepretation:

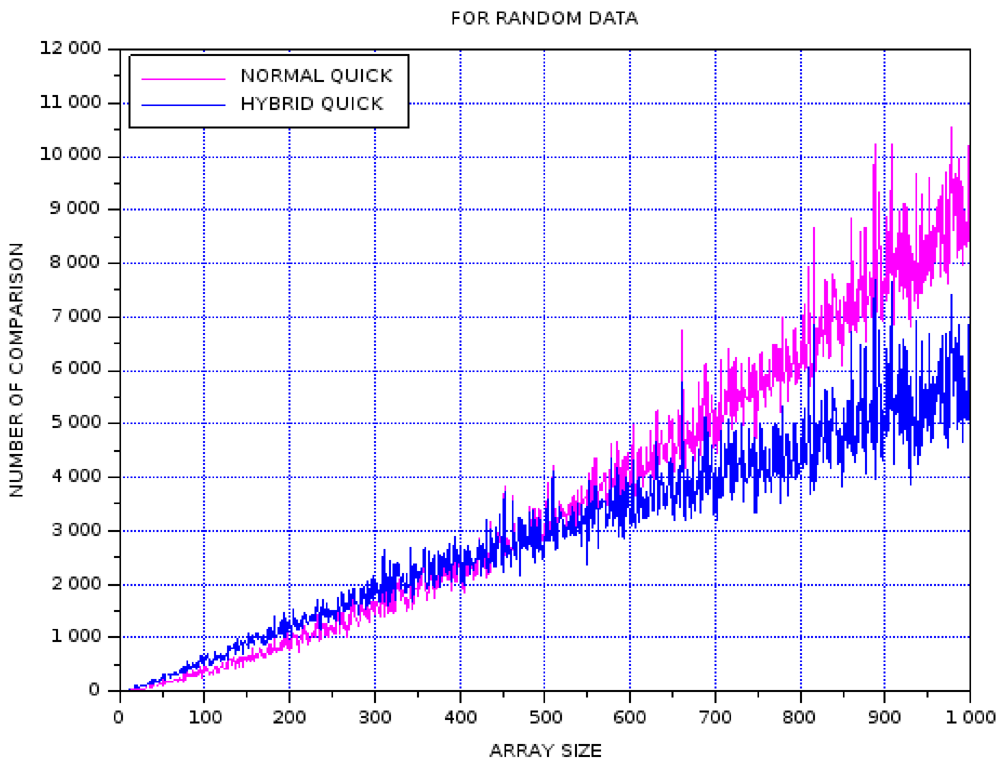


Figure 4.1

4.4 Conclusion:

- From the figure 4.1, we can conclude that both algorithm have similar performance i.e, **O(nlogn)**.
- However, as array size increases, hybrid quick performs best because it then utilizes property of insertion sort.

5 STRASSENS's Matrix multiplication

Practical implementation of Strassens's matrix multiplication algorithm is usually switched to the brute force method after matrix sizes become smaller than some crossover point. Run an experiment to determine such crossover point on your computer system.

5.1 Program Code:

```
//sub function
function[cmpb]= sub(A,B,C,n)
    cmpb=0;
   if n==1 then
        C= A-B;
        cmpb =1;
        return;
    end
    for i=1:n
        for j=1:n
             C(i,j) = A(i,j) - B(i,j);
            cmpb = cmpb+1;
        end
    end
endfunction

//add function

function[cmpa]= add(A,B,C,n)
    cmpa=0;
    if n==1 then
        C= A+B;
        cmpa =1;
        return;
    end
    for i=1:n
        for j=1:n
            C(i,j) = A(i,j) + B(i,j);
            cmpa = cmpa+1;
        end
    end
endfunction

//Strassen function

function[cmp]=strassen_algorithm(A,B,C,n)
```

```
if n == 1 then
    cmp=1;
    return;
else
    n = n/2;
    a11 = zeros(n,n);
    a12 = zeros(n,n);
    a21 = zeros(n,n);
    a22 = zeros(n,n);

    b11 = zeros(n,n);
    b12 = zeros(n,n);
    b21 = zeros(n,n);
    b22 = zeros(n,n);

    p=  zeros(n,n);
    q= zeros(n,n);
    r= zeros(n,n);
    s= zeros(n,n);
    t=  zeros(n,n);
    u=  zeros(n,n);
    v=  zeros(n,n);

    c11 = zeros(n,n);
    c12 =  zeros(n,n);
    c21 = zeros(n,n);
    c22 =  zeros(n,n);

    ares =  zeros(n,n);
    bres =  zeros(n,n);

end
    for i=1:n
        for j=1:n
            a11(i,j) = A(i,j);
            a12(i,j) = A(i,j + n);
            a21(i,j) = A(i + n,j);
            a22(i,j) = A(i + n,j + n);

            b11(i,j) = B(i,j);
            b12(i,j) = B(i,j + n);
            b21(i,j) = B(i + n,j);
            b22(i,j) = B(i + n,j + n);

          end
    end
```

```
        cmpa1 = add(a11,a22,ares,n);
        c= add(b11,b22,bres,n);
        cmpm1 = strassen_algorithm(ares,bres,p,n);

        cmpa2 = add(a21,a22,ares,n);
        cmpm2 = strassen_algorithm(ares,b11,q,n);

        cmpb1 = sub(b12,b22,bres,n);
        cmpm3 = strassen_algorithm(a11,bres,r,n);

        cmpb2 = sub(b21,b11,bres,n);
        cmpm4 = strassen_algorithm(a22,bres,s,n);

        cmpa3 = add(a11,a12,ares,n);
        cmpm5 = strassen_algorithm(ares,b22,t,n);

        cmpb3 = sub(a21,a11,ares,n);
        cmpa4 = add(b11,b12,bres,n);
        cmpm6 = strassen_algorithm(ares,bres,u,n);

        cmpb4 = sub(a12,a22,ares,n);
        cmpa5 = add(b21,b22,bres,n);
        cmpm7 = strassen_algorithm(ares,bres,v,n);

        cmpa=cmpa1+cmpa2+cmpa3+cmpa4+cmpa5+c;
        cmpb = cmpb1+cmpb2+cmpb3+cmpb4;
        cmpm = cmpm1+cmpm2+cmpm3+cmpm4+cmpm5+cmpm6+cmpm7;

        cmp=cmpa+cmpm+cmpb;

endfunction

//main function
cmp = zeros(1,7);
cp = zeros(1,7);
noe = zeros(1,7)

for n =1:7
     k=0;
    noe(n) = n;
    t= 2.^n;
    A = zeros(t,t);
    B= zeros(t,t);
    C= zeros(t,t);
    for i = 1:t
         for j= 1:t
```

```
            for z= 1:t
                k=k+1;
                C(i,j) = A(i,j)*B(i,j);
             end
          end
      end
      cmp(n)=k/100000;
      cp(n)= strassen_algorithm(A,B,C,t)/100000;
end

plot(noe,cmp,"r");
plot(noe,cp);
xgrid(2);
title("PERFORMANCE ANALYSIS OF MATRIX MULTIPLICATION");
xlabel("n {MATRIX SIZE: 2^n}");
ylabel("Number of comparisons in 100000");
legend("NAIVE","STRASSEN");
```

5.2 Conclusion:

The comparisons between Strassen's matrix multiplication algorithm and Brute force matrix multiplication is done using total number of arithmetic operations performed on matrices.
Here are the parameters –

i = Number of iterations
n = 2^i Total number of elements
S(n) = Total number of arithmetic operations on elements of matrices in Strassen's matrix multiplication algorithm with size n
T(n) = Total number of arithmetic operations on elements of matrices in Brute force matrix multiplication algorithm with size n

By Calculating S(n) and T(n) from the algorithms we get,

$$\mathbf{T(n) = O(}n^3\mathbf{);}$$
$$\mathbf{S(n)= 7*S(n/2) + 10*(}n/2^2\mathbf{);}$$

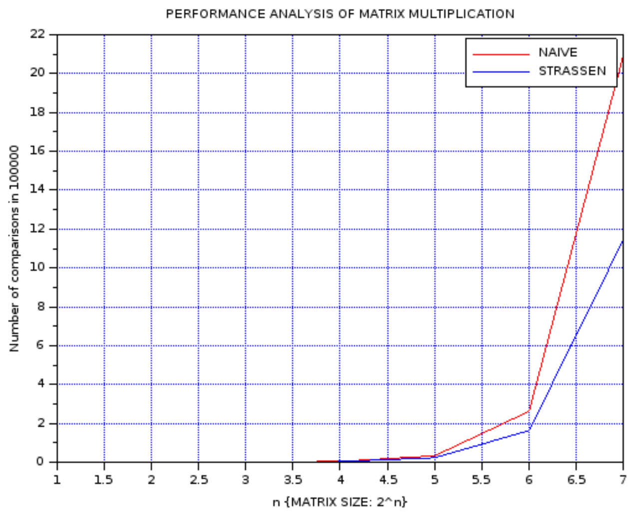


Figure 5.1

Tabulation of the observation is given by Table 5.1 for matrix size ranges from 1 to 128.

MATRIX SIZE	**NUMBER OF COMPARISON**	
	STRASSEN's	**NAIVE**
1 * 1	1	1
2 * 2	9.12	8
4 * 4	73	64
8 * 8	689	512
16 * 16	2450	4096.
32 * 32	17001	32768
64 * 64	120012	262144
128 * 128	834143	2097152

Table 5.1

6 QUICK SELECT

Use the QUICK SELECT algorithm to find 3 rd 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.

6.1 Program Code:

```
// Standard partition process of QuickSort().
// It considers the last element as pivot
// and moves all larger element to left of
// it and smaller elements to right
function[A,index,cp] = partition(A,l,r,c)
    cp=c;
    x= A(r);
    i=1;
    for j=1:r-1
        if A(j) >=x then
            cp=cp+1;
            t= A(i);
            A(i)=A(j)
            A(j)=t;
            i=i+1;
        end
    end
    t=A(i);
    A(i)=A(r);
    A(r)=t;
    index =i;
endfunction

// This function returns k'th largest
// element in A[l..r] using QuickSort
// based method.
function[A,index,cp] = kthsmall(A,l,r,k,c)
    cp=c;
    if (k>0 && k<= r-l+1) then
        [A,index,cp] = partition(A,l,r,cp);
        if( index-l == k -1)then
            return;
        end
        if  (index -l > k-1) then
            cp=cp+1;
            [A,index,cp]=   kthsmall(A,l,index-1,k,cp);
        end
```

```
        [A, index , cp ]  =  kthsmall(A, index+1,r , k−index+l −1,cp );
    end
endfunction
//main  method

noe  =  zeros(1 ,450);
cmp =  zeros(1 ,450);
c=0;
i=1;
for  n=  50:500
    noe(i)  =n;
    A=  round(rand(1 ,n)∗100);
    l=1;
    r=n;
    k=3;
   [A, index , cp ]  =  kthsmall(A, l , r , k , c );
   cmp(i)  =cp;
   i=i+1;
end
plot(noe ,cmp,"m");
xgrid;
title("FINDING  3rd  LARGEST ELEMENT  USING  QUICK  SELECT");
xlabel("ARRAY  SIZE");
ylabel("NUMBER  OF  COMPARISON");
```

6.2 Conclusion:

In this experiment, Array with random values are chosen. Due to this, There is a considerable fluctuations in number of comparisons.

- The number of comparison to find 3rd largest element using quick-select varies with array size(as size increases, comparison also increases) and also by other factors.
- From this experiment, It is observed that number of comparison increases when input array is almost in decreasing order.
- And also when elements are repeating.

Tabulation of the observation is given by Table 6.1

ARRAY SIZE	NUMBER OF COMPARISON
200	653.
201	188.
202	360.
203	461.
204	437.
205	496.
206	438.
207	259.
208	158.
209	427.
210	243.

Table 6.1: **TABULATION FOR ARRAY SIZE FROM 200 TO 210**

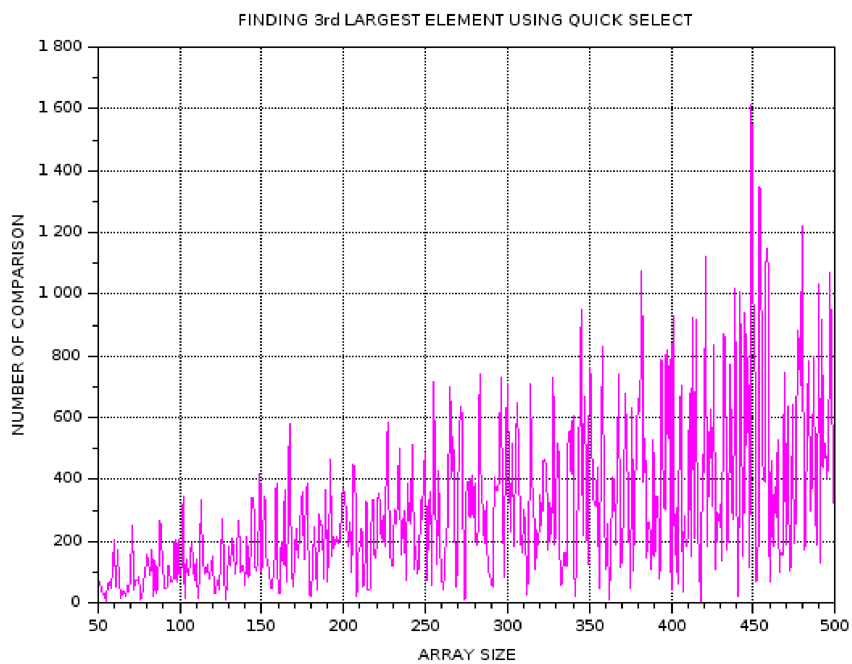


Figure 6.1

7 Iterative Binary Search

Write programs to implement recursive and iterative versions of binary search and compare the performance. For a appropriate size of n=20(say), have each algorithm find every element in the set. Then try all n+1 possible unsuccessful search. The performance, of these versions of binary search are to be reported graphically with your observations.

7.1 Program Code:

```
//Recursive binary search algorithm
function[c,i,t] = recursive_binary(a,l,h,k,c,time)
tic();
mid=round((l+h)/2);
i=-1;
c=c+1;
    if l<=h then
       if k== a(mid) then
           i=mid
           break;
       elseif k<a(mid)
                 [c,i]=recursive_binary(a,l,mid-1,k,c);
            else
                 [c,i]=recursive_binary(a,mid+1,h,k,c);
             end
  end
     t=time+toc();
endfunction

//Iterative binary search algorithm
function[icp,in,t]=iterative_binary(A,l,h,k,icp,itime)
     tic();
     in=-1;
     while (l<=h)
         icp = icp+1;
         mid=round((l+h)/2);
         if ( A(mid) == k)
             in=mid;
             break;
         end
         if (A(mid) < k)
             l=mid+1;
         else
             h=mid-1;
         end
     end
```

```
    t=itime+toc();
endfunction

//main program

s=1;
st =zeros(1,10);
ust = zeros(1,10);
cmp=zeros(1,10);
ucmp=zeros(1,10);

ist =zeros(1,10);
iust = zeros(1,10);
icmp=zeros(1,10);
iucmp=zeros(1,10);

noe =zeros(1,10);
for n=10:10:100
    rec = zeros(1,n);
    noe(s)=n;
    cp=0;
    ucp=0;
    time=0;
    utime=0;

    icp=0;
    iucp=0;
    itime=0;
    iutime=0;
    A= round(rand(1,n)*100);
    A=gsort(A,"g","i");
    B= round(rand(1,n+1)*100+100);
        for i=1:n
            k = A(i);
            [cp,index,time]=
            recursive_binary(A,1,n,k,cp,time);
            [icp,in,itime] =
            iterative_binary(A,1,n,k,icp,itime);

        end
        cmp(s) = cp;
        icmp(s)=icp;
        st(s)=time*1000;
        ist(s)=itime*1000;
        for i=1:n+1
            l=B(i);
```

```
            [ucp, index, utime]=
            recursive_binary(A,1,n,l,ucp,utime);
             [iucp, in, iutime] = iterative_binary(A,1,n,k,
             iucp, iutime);
        end
        iucmp(s)=iucp;
        ucmp(s)=ucp;
        ust(s)=utime*1000;
        iust(s)=iutime*1000;
        s=s+1;
end
plot(noe, st, "m");
plot(noe, ist, "k");
xlabel("ARRAY SIZE");
ylabel("TIME TAKEN IN ms");
title("SUCCESSFUL SEARCH");
xgrid(2);
legend("RECURSIVE","ITERATIVE",2);
```

7.2 Interpretation:

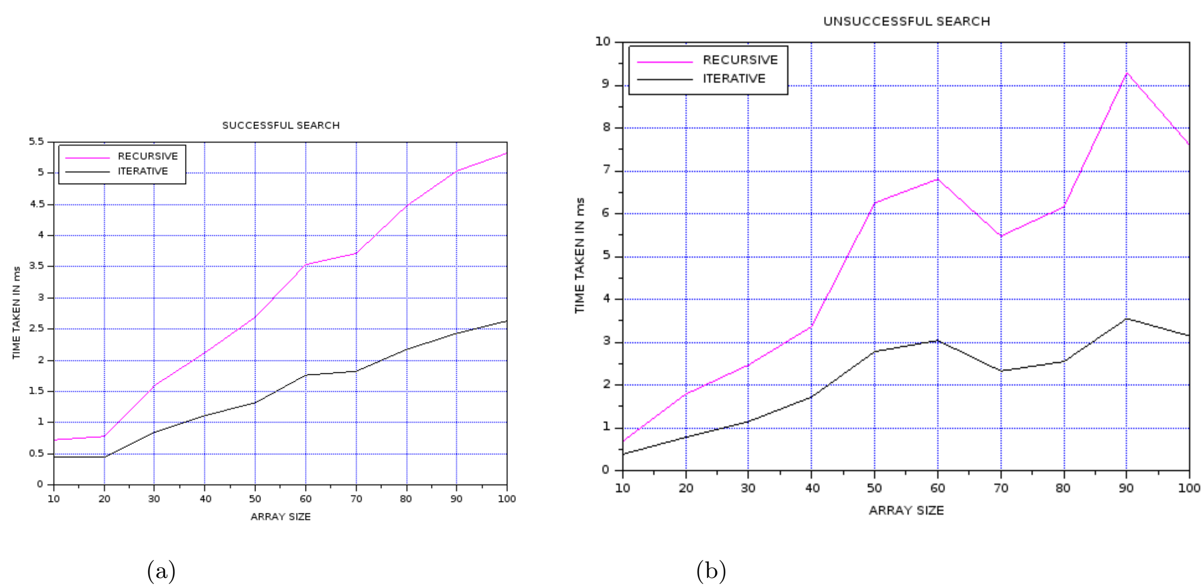
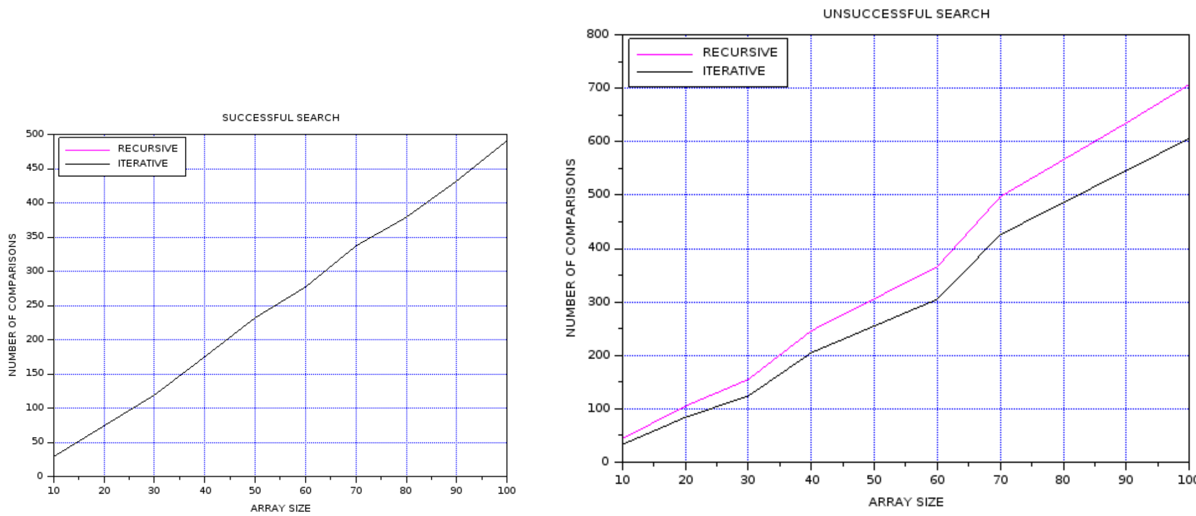
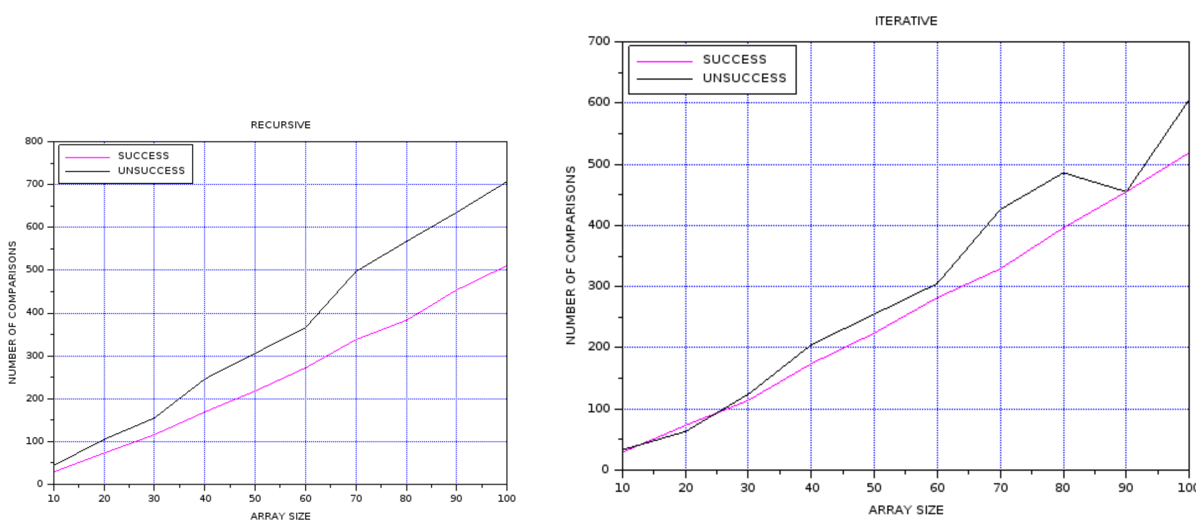


Figure 7.1: Comparison of time taken by both algorithms



(a) (b)

Figure 7.2: Number of times elements are compared in both algorithms



(a) (b)

Figure 7.3: Comparison of successful and unsuccessful search in both algorithms

7.3 Comments:

- From figure 7.1b and 7.1a , we can conclude than recursion takes more time than iteration (Recursion is more expensive operation than a simple loop iteration).
  Tabulation of the observation is given by Table 7.1:

MATRIX SIZE	SUCCESSFUL SEARCH		UNSUCCESSFUL SEARCH	
	RECURSIVE	ITERATIVE	RECURSIVE	ITERATIVE
10	0.843	0.549	0.613	0.292
20	1.412	0.793	1.916	0.724
30	1.611	0.892	1.687	0.728
40	2.01	1.026	2.679	1.157
50	2.541	1.287	5.236	2.409
60	5.47	2.674	6.157	2.605
70	4.583	2.341	7.505	3.376
80	4.649	2.258	6.273	2.506
90	5.175	2.524	7.027	2.515
100	5.88	2.832	7.797	3.137

Table 7.1: TIME TAKEN

- Both iterative and recursive binary search have same number of comparisons and same time complexity –**O(logn)**. This can be observed from figure 7.2.
  In figure 7.2a, for successful search both algorithm takes same number of comparisons whereas in unsuccessful - number of comparison is not same, recursion takes more comparison (figure7.2b).
- Since we are trying n+1 unsuccessful search, number of comparison for unsuccessful search in both algorithm is greater than successful, it is shown in figure 7.3.

8 Matrix Chain Multiplication Comparison

Given a matrix chain A 1 ... A n with the dimension of each of the matrices given by the vector p = ¡12,21,65,18,24,93,121,16,41,31,47,5,47,29,76,18,72,15¿. (n=17) Write and run both the dynamic programming and memorized versions of this algorithm to determine the minimum number of multiplications that are needed (use type longint) and the factorization that produces this best case number of multiplications. Run each of the two programs over an appropriately large number of times (put each in a loop to run repeatedly x times) and obtain the times at the beginning and end of the run. Use these times to determine the comparative runtimes of the two algorithms.

8.1 Program Code:

```
function[t,res]= dynamic(arr,n)
     tic();
   m= zeros(n,n);
    for l=2:n-1
         for i=2:n-l+1
              j=i+l-1;
              if j == n+1 then
                   continue;
              end
              m(i,j)=1000000000;//this value is taken as maximum
              integr value in this program
              for k=i:j-1
                   q= m(i,k)+m(k+1,j)+arr(i-1)*arr(k)*arr(j);
                   if q<m(i,j) then
                       m(i,j)=q;
                   end
               end
           end
         end
         res=m(2,n);
         t=toc();
         return;
endfunction
function[time,t] = memoised(arr,i,j,dp)
    tic();
     if i==j then
          t=0;
          return;
     end
     if dp(i,j) ~=0 then
          t=dp(i,j);
          return;
```

```
    end
    dp(i,j)=10000000;
    for k=i:j-1
        [t1 ,res1]=memoised(arr,i,k,dp);
        [t2,res2]= memoised(arr,k+1,j,dp);
        dp(i,j)=min(dp(i,j), res1 +res2 + arr(i-1)*
        arr(k)*arr(j));
    end
    t=dp(i,j);
    time=toc();
    return;
endfunction
arr =[12,21,65,18,24,93,121,16,41,31,47,5,47,29,
76,18,72,15];
ti=zeros(1,100);
tim=zeros(1,100);
n=length(arr);
dp=zeros(n,n);
for i=1:100
    [mtime,t2]= memoised(arr,2,n,dp);
    [time,t1]=dynamic(arr,n);
    ti(i)=time *1000000;
    tim(i)=mtime*1000000;
end
clf;
printf("Minimum number of multiplications using
tabulation method is %d ",t1);
printf("Minimum number of multiplications using
memoisation method is %d ",t2);
plot(ti,"m");
plot(tim,"k");
xlabel("NUMBER OF TIMES ALGORITHM IS EXECUTED");
ylabel("TIME TAKEN IN ns");
title("MATRIX CHAIN MULTIPLICATION");
xgrid(2);
legend("TABULATION","MEMOISED");
```

8.2 Conclusion:

Figure 8.2 shows the comparison between Dynamic bottom up and Top down memorized matrix chain multiplication with respect to time in milliseconds. The graph shows that bottom up method performs way better than the memorized version. The fundamental reason is that the memorized version uses two recursive branches inside the loop while dynamic version uses three simple loops, which makes the former way more expensive. The time taken by memorized

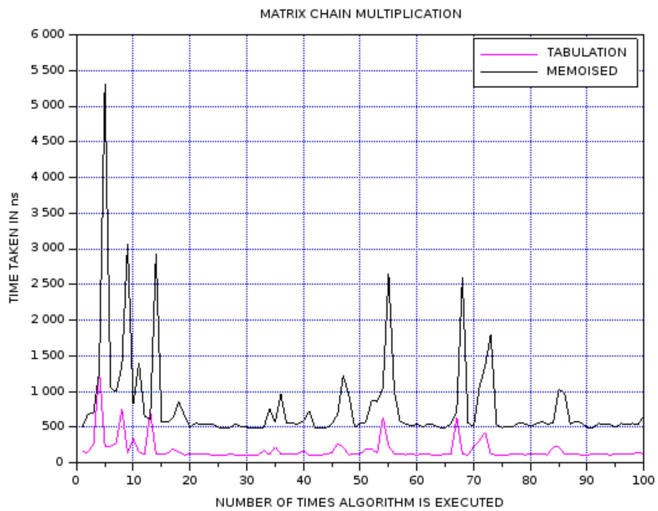


Figure 8.2

version is approximately double that of dynamic version.

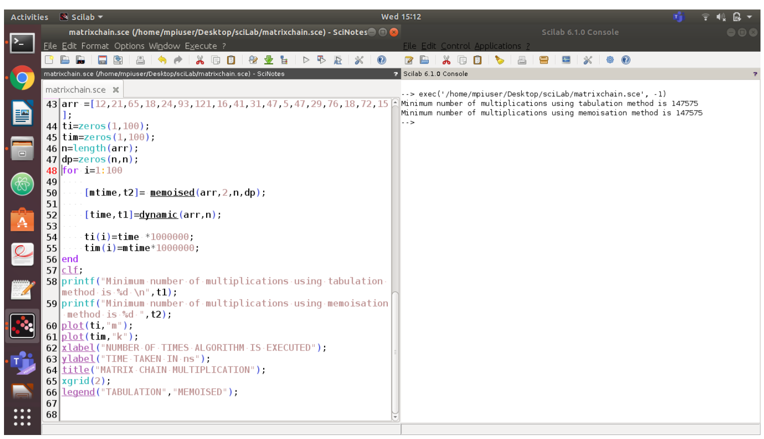


Figure 8.1

Tabulation of the observation for first 20 iterations is given by Table8.1

NUMBER OF TIMES ALGORITHM EXECUTED	TIME TAKEN IN seconds	
	TABULATION	**MEMOIZATION**
1	0.00013	0.000497
2	0.000193	0.000629
3	0.000204	0.000895
4	0.000196	0.000937
5	0.000212	0.000916
6	0.000199	0.000956
7	0.000126	0.000926
8	0.000183	0.000997
9	0.000096	0.00079
10	0.000096	0.000442
11	0.000096	0.000464
12	0.000095	0.000444
13	0.000094	0.000438
14	0.000095	0.000434
15	0.000094	0.000439
16	0.000109	0.000442
17	0.000098	0.000494
18	0.000108	0.000457
19	0.000108	0.000499
20	0.000107	0.000478

Table 8.1

9 BINOMIAL COEFFICIENTS

Computing the **binomial coefficients** C(n, k) defined by the following recursive formula:

$$C(n,k) = \begin{cases} 1 & k = 0 or k = n; \\ C(n-1,k) + C(n-1,k-1) & 0 < k < n; \\ 0 & otherwise \end{cases}$$

Write the program with three different algorithm to compute binomial coefficients C(n, k) and compare them?

9.1 Program Code:

```
//recursive approach
function[time,C] = recursive_bin(i,k,time)
        tic();
        t1=0;
        t2=0;
    if (k==0 || i==k) then
         C=1
         time = time+toc();
         return ;
    elseif (k >i ) then
         C=0;
          time = time+toc();
          return ;
         end
   [t1,C1]=recursive_bin(i-1,k,t1);
   [t2,C2]=recursive_bin(i-1,k-1,t2);
   C=C1+C2;
   time = t1+t2+time+toc();
   return;
endfunction

//dynamic approach
function [dtime,t]=dynamic_binomial(i,k,dtime)
    tic();
    C=zeros(i+1,k+1);
    for l=1:i+1
        for j=1:min(l,k)+1
            if (j==l || l==1 ||j==1 )then
              C(l,j) =1;
            else
                C(l,j)=C(l-1,j-1)+C(l-1,j);
```

```
            end
        end
    end
    t=C(i+1,k+1);
    dtime = dtime+toc();
endfunction

//formula_based approach
function [time] = formula_based(i,k,time)
    tic();
    if (k==0 || i==k )then
        res=1;
        time=time+toc();
        return;
    elseif (k<i) then
        num_p=1;
        den_p=1;
        if (i< 2* k) then
            k=i-k;
         for j = 1:k+1
             num_p = num_p * (i+1 -j);
             den_p =den_p *j;
         end
         res =num_p/den_p;
         time=time+toc();
         return;
        end
    end
    res=0;
    time=time+toc();
    return;
endfunction


//main program
n=10;
r=100;
s=1;
ft=zeros(1,n)
rt=zeros(1,n);
dt=zeros(1,n)
noe=zeros(1,n);
for i=1:n
    time=0;
    dtime=0;
    ftime=0;
```

```
    noe(s)=n;
    for j=1:r
        k=grand(1,1,"uin",0,i);
        [time]= recursive_bin(i,k,time);
        [dtime]=dynamic_binomial(i,k,dtime);
        [ftime]=formula_based(i,k,ftime);
    end
    rt(s)=(time/100);
    dt(s)=(dtime/100);
    ft(s)=ftime/100;
    s=s+1;
end
clf
plot(rt,"k");
plot(dt,"m");
plot(ft,"r")
xlabel("N");
ylabel("TIME TAKEN IN s");
title("VARIANTS OF BINARY SEARCH");
xgrid(2);
legend("RECURSIVE","DYNAMIC","FORMULA")
```

9.2 Conclusion:

Recursion Algorithm:
The recursive algorithm uses the recursive equation directly to compute the binomial coefficient off the given numbers n, k. This makes the algorithm very inefficient with time complexity of O($2^n/\sqrt{n}$)using Stirling's approximation.

Dynamic Algorithm:
The dynamic algorithm uses a table of memory size (n + 1) (k + 1) (including n = 0 and k = 0) to store the result of intermediate binomial coefficients which will be used to compute the final binomial coefficient.

The table is initially saved with values with trivial cases. The time complexity is of order n 2 . One of greatest disadvantage is that half of the memory used in table is wasted due to the restriction k $\leq n$.

Formula based algorithm:
The third algorithm simplifies the binomial coefficient formula in the following way-

$$n_{C_r} = n!/k! * (n-k)!$$

is simplified to

$$n_{C_r} = \Pi_{i=1}^{k}(n+1-i)/\Pi_{i=1}^{k} i$$

This makes the complexity of the algorithm O(k).
This makes the algorithm way faster than both the above methods with using only two variables for storing the numerator and the denominator separately.

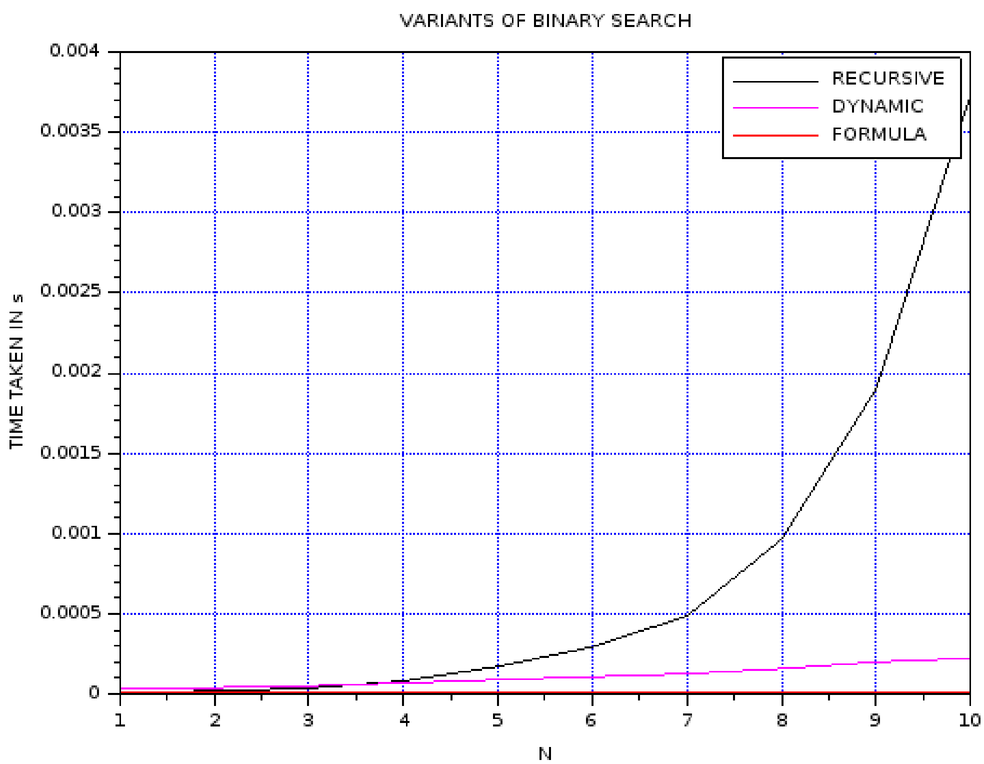


Figure 9.1

Tabulation of the observation is given by table9.1

N	**TIME TAKEN IN seconds**		
	RECURSIVE	**DYNAMIC**	**FORMULA**
1	0.0000205	0.0000244	0.0000127
2	0.0000296	0.0000237	0.000013
3	0.0000544	0.0000236	0.0000105
4	0.0000866	0.0000227	0.0000108
5	0.0001725	0.0000264	0.000013
6	0.0002637	0.0000339	0.0000138
7	0.0005415	0.0000355	0.0000178
8	0.0010745	0.0000392	0.0000199
9	0.0021787	0.0000544	0.0000161
10	0.0036043	0.0000574	0.0000164

Table 9.1

10 0-1 KNAPSACK PROBLEM

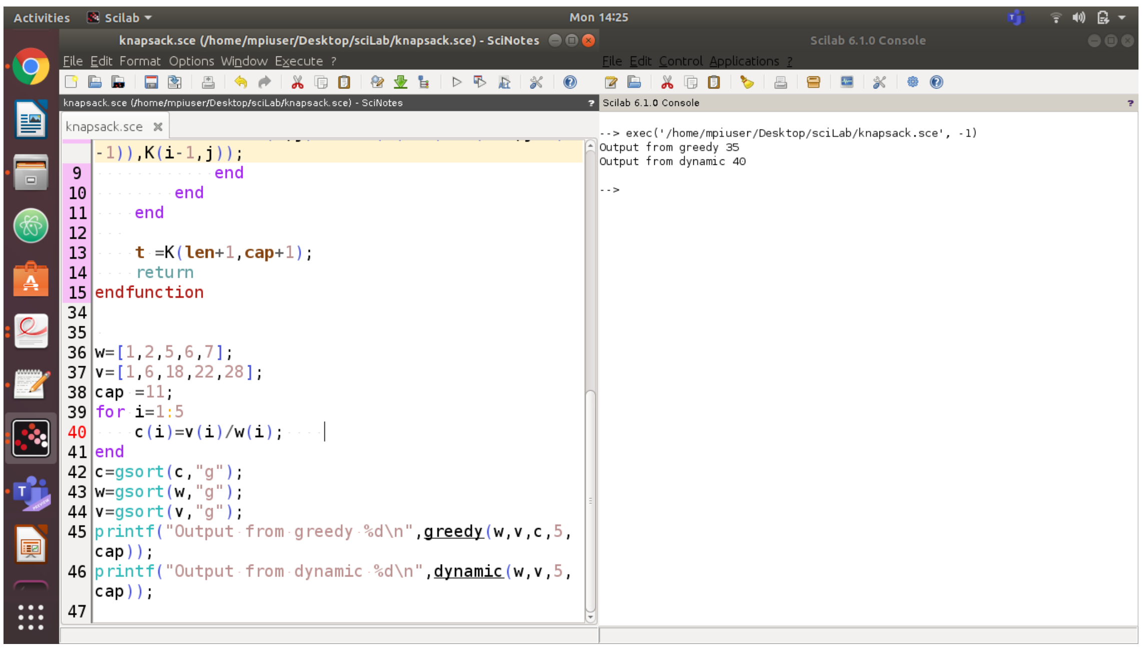
Write a program that computes optimal solution to the 0–1 Knapsack Problem using dynamic programming? You may test your program with the following example:
There are n = 5 objects with integer weights w[1..5] = 1,2,5,6,7, and values v[1..5] = 1,6,18,22,28. Assuming a knapsack capacity of 11).
Compare your solution to a greedy algorithm that computes sub-optimal solution the 0–1 Knapsack Problem.

10.1 Program Code:

```
//greedy approach
function[t_v] = greedy(w,v,c,len,cap)
    t_v=0;
    w_l=cap;
    for i=1:len
        if  w_l >= w(i)then
            t_v= t_v+v(i);
            w_l = w_l- w(i);
        end
        if w_l <=0 then
            break;
        end
    end
   return;
endfunction
//dynamic approach
function[ t ]=dynamic(w,v,len,cap)
    K=zeros(len+1,cap+1);
    for i=1:len+1
        for j=1:cap+1
            if (i == 1 || j ==1 )
                K(i,j) =0;
            elseif ( j-w(i-1) >= 1)
                K(i,j) = max(v(i-1) +K(i-1,j-w(i-1)),K(i-1,j));
            end
        end
    end
    t =K(len+1,cap+1);
    return
endfunction
//main program
w=[1,2,5,6,7];
v=[1,6,18,22,28];
cap =11;
```

```
for i=1:5
    c(i)=v(i)/w(i);
end
c=gsort(c,"g");
w=gsort(w,"g");
v=gsort(v,"g");
printf("Output from greedy %d\n",greedy(w,v,c,5,cap));
printf("Output from dynamic %d\n",dynamic(w,v,5,cap));
```

10.2 Result:



10.3 Comments:

For Dynamic programming:
The Solution to knapsack problem can be stated as –

T[i,W] = max(T[i-1,W-w_i]+v_i, T[i-1,W] ....if $w_i \leq W$
and T[i,W]= T[i-1,W].......Otherwise

Where,
T[i, W] = Total value gained with W weight and i items remaining
w_i , V_i = Weight and Value of i^{th} item respectively

The solution space of this problem is 2^n where n is the number of items as each item can either be picked or not.
Using dynamic problem an array T[i, W] can be used to save the intermediate results. The time complexity is **O(n^2)**

For greedy algorithm:

$$T[i,W] = T[i\text{-}1,W\text{-}w_i]+v_i \text{..........if } w_i \leq W$$
$$\text{and } T[i,W] = T[i\text{-}1,W]\text{.........Otherwise.}$$

The i can be determined by sorting the array of V_i/w_i in descending order and then iterating the sorted array from largest to smallest. Complexity of greedy algorithm is **O(nlogn)**

11 STRING MATCHING ALGORITHM

Given two strings P and T over the same alphabet set , determine whether P occurs as a substring in T (or find in which position(s) P occurs as a substring in T). The strings P and T are called pattern and text respectively. Compare the efficiency of three string matching algorithms (Brute-Force Algorithm, Knuth-Morris-Pratt and Boyer-Moore Algorithm) by varying pattern length [1-15] for n=5000.

11.1 Program Code:

```
function[time,cp]= naive_matching(pattern,str)
    tic();
    cp=0;
    m=length(pattern);
    n=length(str);
    indices=[];//it gives indices at which
    pattern occurs(starting point)
    k=1;
    count=0;//it gives how many time pattern occurs
    for i=1:n-m+1
        for j=1:m
            t=i+j-1;
            cp=cp+1;
           if (pattern(j)  ~= str(i+j-1)) then
               break;
           end
           if (j==m) then
               count=count+1;
              indices(k)=i;
              k=k+1;
           end
        end
    end
    time=toc();
  return;
endfunction
function [lps,cp] = compute_lps(pattern,m,lps,cp)
    len=1;
    lps(1)=1;
    i=2;
    while( i <m+1)
        cp=cp+1;
        if pattern(i) == pattern(len) then
            len=len+1;
            lps(i)=len;
```

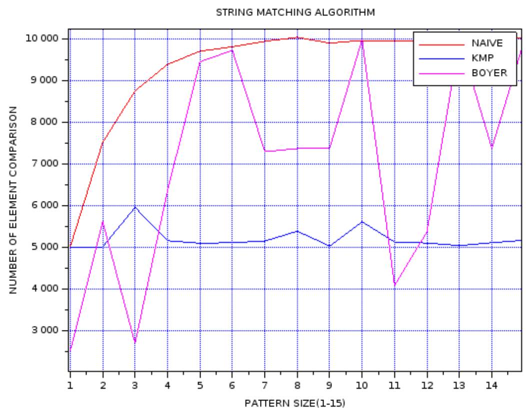
```
        i=i+1;
    else
        if len ~= 1 then
            len= lps(len-1);
        else
            lps(i)=1;
            i=i+1;
        end
    end
  end
endfunction

function [last]=compute_last(pattern)
  last=zeros(1,4);
  for i=1:4
      last(i)=-1;
  end
  for j=length(pattern):-1:1
      t= pattern(j);
      if last(t) == -1 then
          last(t)=j;
      end
  end
endfunction

function[time,cp]= boyer_moore(pattern,str)
  tic();
  t=[];
  cp=0;
  k=1;
  m=length(pattern);
  n=length(str);
  [last]= compute_last(pattern);
   i=m;
   j=m;
   while (i <=n)
       cp=cp+1;
       if pattern(j) == str(i) then
           if j==1 then
               t(k)=i;
               k=k+1;
               i =i+1+ m;
              j = m;
           else
               i=i-1;
               j=j-1;
```

```
        end
    else
        i = i + m - min(j, 1 + last(str(i)))+1;
        j=m;
    end
  end
  time=toc();
endfunction

function[time,cp] = kmp_search(pattern,str)
    tic();
    cp=0;
    n=length(str);
    m=length(pattern);
    lps= zeros(1,m);
    [lps,cp]= compute_lps(pattern,m,lps,cp);
    i=1;
    j=1;
    while(i < n+1)
        cp=cp+1;
        if pattern(j) == str(i) then
            i=i+1;
            j=j+1;
        end
        if j == m+1 then
            //printf("Found pattern at index %d \n", i - j+1);
            j=lps(j-1);
        elseif i<n+1 && pattern(j) ~= str(i) then
                if j~=1 then
                    j=lps(j-1);
                else
                     i =i+1;
                end
        end
    end
    time=toc();
endfunction

function[s]= get_string(n,alpha)
    s=zeros(1,n);
    for i =1:n
        s(i)=alpha(grand(1,1,"uin",1,4));
    end
    return
endfunction
```

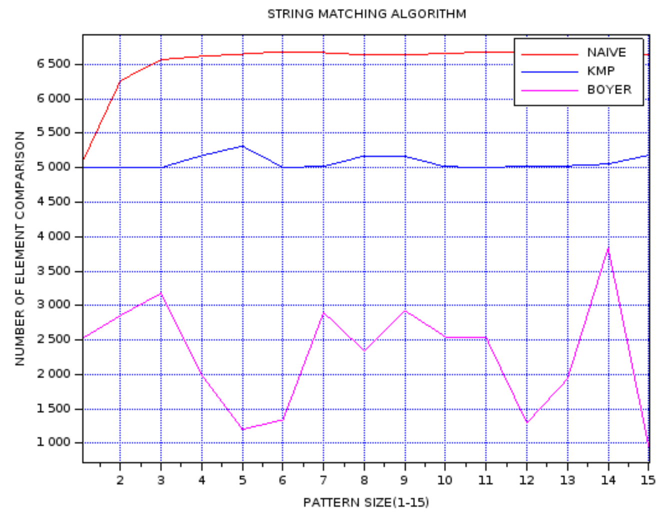
```
alpha= grand(1, "prm", 1:4);
n=5000;
m=15;

noe = zeros(1,m);
cmp = zeros(1,m);
cmpk = zeros(1,m);
cmpb = zeros(1,m);
    str = get_string(n,alpha);
    for j= 1:m
        noe(j)=j;
        pattern= get_string(j,alpha);
        [time,cp]=naive_matching(pattern,str);
        [ktime,cpk]=kmp_search(pattern,str);
        [btime,cpb]=boyer_moore(pattern,str)
        cmp(j)=time
        cmpk(j)=ktime;
        cmpb(j)=btime;
    end
clf;
plot(noe,cmp,"r");
plot(noe,cmpk);
plot(noe,cmpb,"m");
xgrid(2);
xlabel("PATTERN SIZE(1-15)");
ylabel("NUMBER OF ELEMENT COMPARISON");
title("STRING MATCHING ALGORITHM");
legend("NAIVE ","KMP ","BOYER");
```

11.2 Conclusion:

Boyer-Moore algorithm is extremely fast on large alphabet (relative to the length of the pattern). The payoff is not as for binary strings or for very short patterns. For binary strings Knuth-Morris-Pratt algorithm is recommended. For the very shortest patterns, the naïve algorithm may be better.



(a) Here, alphabet size = 2



(b) Here, alphabet size = 4

12 MATRIX CHAIN MULTIPLICATION

Write a program to compute the best ordering of matrix multiplication. Include the routine to print the actual ordering.

12.1 Program Code:

```
//recursive function to print the ordering
function[name] = printPara(i,j,br,name)
    if i==j then
        printf(name);
        t=ascii(name);
        name = char(t+1);
        return;
    end
    printf("(");
    [name]=printPara(i,br(i,j),br,name);
    [name]=printPara(br(i,j)+1,j,br,name);
    printf(")");
endfunction

function matrixChainOrder(arr,n)
   m= zeros(n,n);
   br=zeros(n,n);
   for l=2:n-1
       for i=2:n-l+1
           j=i+l-1;
           m(i,j)=1000000000;//this value is
           taken as maximum integr value in this program
           for k=i:j-1
               q= m(i,k)+m(k+1,j)+arr(i-1)*arr(k)*arr(j);
               if q<m(i,j) then
                   m(i,j)=q;
                   br(i,j)=k;
               end
           end
       end
   end
   name = char('A');
   printf("\n");
   printf("Optimal Parenthesization is : ");
   printPara(2,n,br,name);
   printf("\nOptimal Cost is :%d\n",m(2,n));
   return;
endfunction
```

```
arr=[12,21,65,18,24,93,121,16,41,31,47,5,47,29,76,18,72,15];
n=length(arr);
matrixChainOrder(arr,n);
```

12.2 Result:

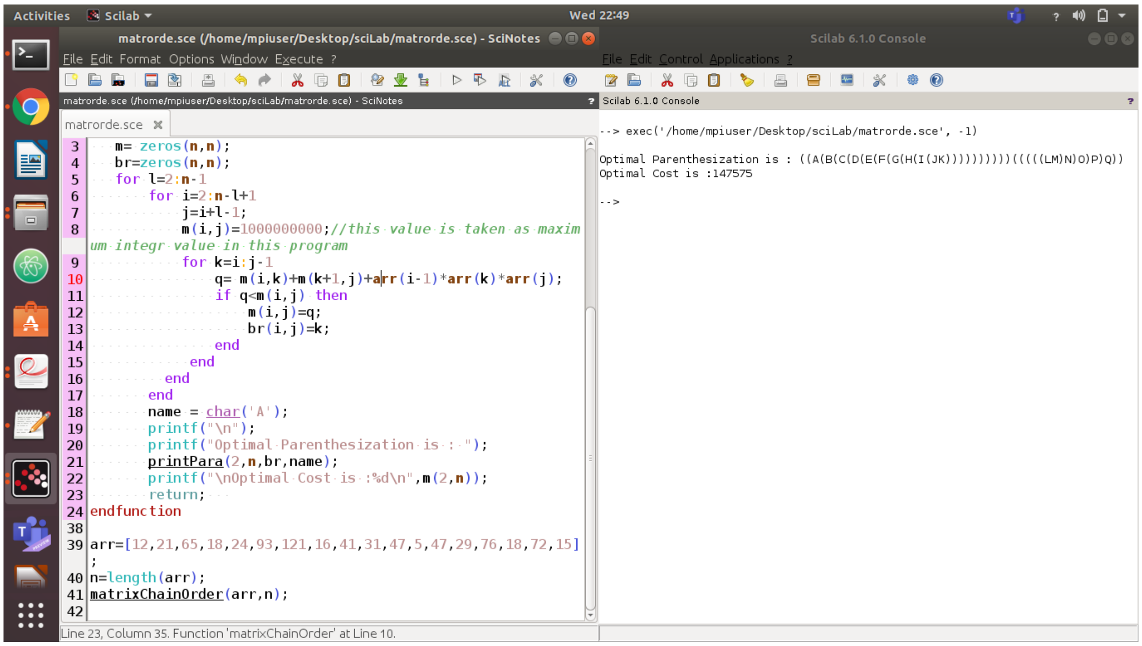


Figure 12.1

13 SPANNING TREE

Write a program obtain minimum cost spanning tree the above NSF network using Prim's algorithm, Kruskal's algorithm and Boruvka's algorithm. Use the appropriate data structure to store the computed spanning tree for another application (broadcasting a message to all nodes from any source node).

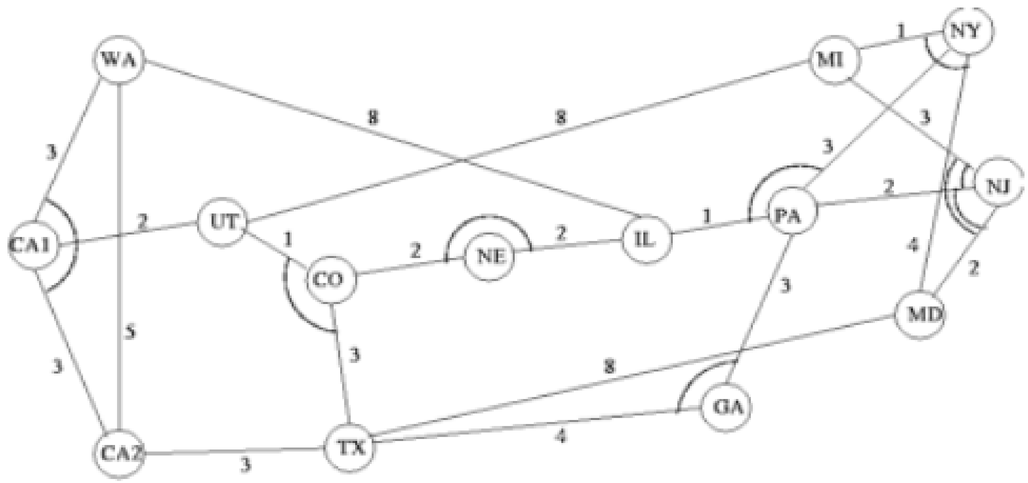


Figure 13.1

13.1 Program Code:

****

```
//KRUSKAL METHOD

function [c] = connected(X)
c = 0;
i = 1;
j = 2;
a = size(X);
while(j > i)
    if X(i,j) == 0
        j = j + 1;
    else
        X(i,:) = X(i,:) | X(j,:);
        X(:,i) = X(:,i) | X(:,j);
        X(j,:) = [];
        X(:,j) = [];
        a(1) = a(1) - 1;
    end
```

```
    if (j > a(1)) & (i < a(1))
        j = i + 2;
        i = i + 1;
        c = 1;
        break
    else
        if i >= a(1)
            i = j;
        end
    end
end
endfunction

function [korif,c]=iscircle(korif,akmi)
    g=max(korif)+1;
    c=0;
    n=length(korif);
    if korif(akmi(1))==0 & korif(akmi(2))==0
        korif(akmi(1))=g;
        korif(akmi(2))=g;
    elseif korif(akmi(1))==0
        korif(akmi(1))=korif(akmi(2));
    elseif korif(akmi(2))==0
        korif(akmi(2))=korif(akmi(1));
    elseif korif(akmi(1))==korif(akmi(2))
        c=1;
        return
    else
        m=max(korif(akmi(1)),korif(akmi(2)));
        for i=1:n
            if korif(i)==m
            korif(i)=min(korif(akmi(1)),korif(akmi
            (2)));
       end
   end
end
endfunction

function A = ascWeightBubb(A,col)
[r c] = size(A);

for i = 1 : r - 1
    d = r + 1 - i;
    for j = 1 : d - 1
        if A(j,col) > A(j + 1,col)
```

```
            A([j j + 1],:) = A([j + 1 j],:);
        end
    end
end
endfunction

function [w,T] = kruskal(PV)
   row = size(PV,1);
    X = [];
//create graph's adjacency matrix
    for i = 1 : row
        X(PV(i,1),PV(i,2)) = 1;
        X(PV(i,2),PV(i,1)) = 1;
    end
    n = size(X,1);
//check if graph is connected
    con = connected(X);
    if con == 1
    error('Graph is not connected');
    end
//sort PV by ascending weights order, here
    bubblesort is used
    PV = ascWeightBubb(PV,3);
    korif = zeros(1,n);
    T = zeros(n);
    for i = 1 : row

    akmi = PV(i,[1 2]);
    [korif,c] = iscircle(korif,akmi);
    if c == 1
       PV(i,:) = [0 0 0];
   end
end

w = sum(PV(:,3)');

for i = 1 : row
    if PV(i,[1 2]) ~= [0 0]
        T(PV(i,1),PV(i,2)) = 1;
        T(PV(i,2),PV(i,1)) = 1;
    end
end
endfunction

PV=[1,2, 3; 1,3,5;1,8,8;2,3,3;2,4,2;3,
6,3;4,11,8;4,5,1;5,6,3;5,7,2;6,10,4;6,13,
```

```
8;7,8,2;8,9,1;9,12,3;9,14,2;9,10,3;11,12,1
;11,14,3;12,13,4;13,14,2];
 [w T] = kruskal(PV);
 printf("Edges present in minimum spanning tree:\n")
 for i=1:14
     for j=i:14
         if T(i,j) ==1 then

             printf("(%d,%d)\n",i,j);
         end
     end
 end
 printf("\n");
 printf("Minimum cost of the spanning tree:%d",w);
```

****

13.2 Conclusion:

The vertices of the graph in the figure 13.1 is indexed according to the Table 13.1

Index	**Vertex**
1	WA
2	CA1
3	CA2
4	UT
5	CO
6	TX
7	NE
8	IL
9	PA
10	GA
11	M1
12	NY
13	MD
14	NJ

Table 13.1

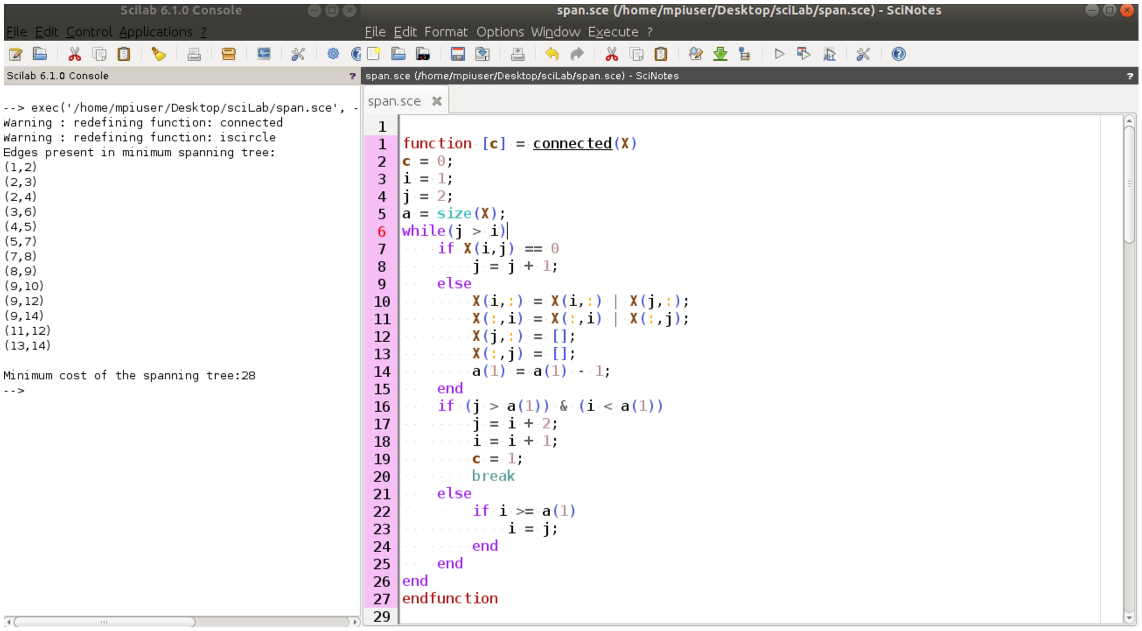


Figure 13.2: KRUSKAL METHOD OUTPUT