122CS0067 Amiya Chowdhury

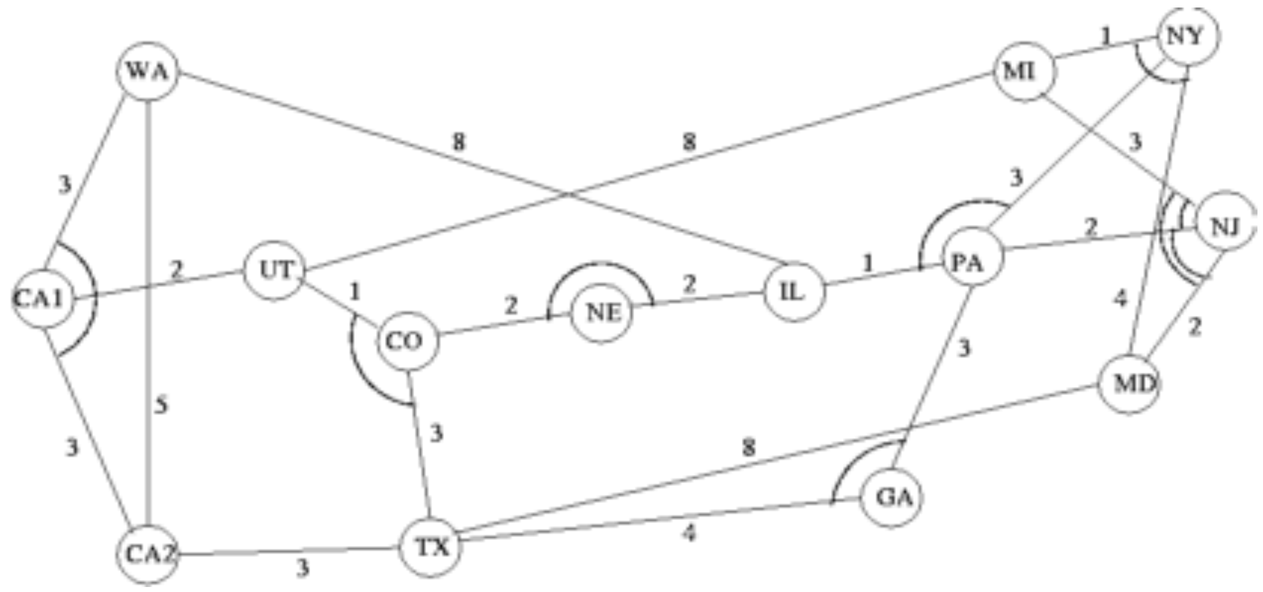
Q.11. STRASSENS's Matrix multiplication

Practical implementation of Strassens's matrix multiplication algorithm usually switch 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.

Q.12. 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*.

Q.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.

Q.14. 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 to be reported graphically with your observations.

Q.15. Iterative MergeSort and QuickSort

Compare the performance of the iterative version of MergeSort & QuickSort ?

122CS0067 Amiya Chowdhury

Q.11. STRASSENS's Matrix multiplication

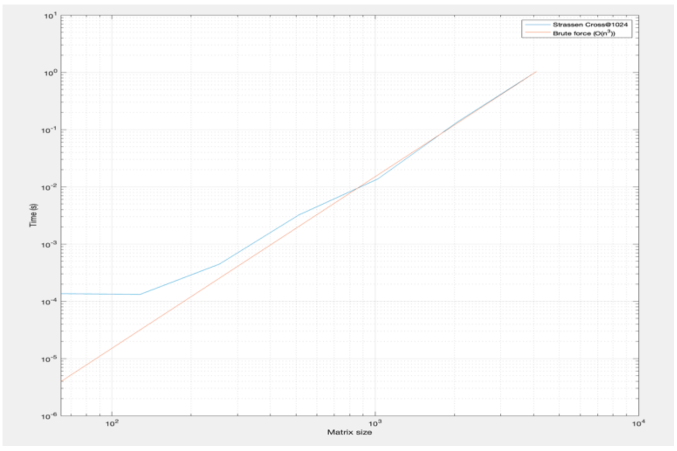
Practical implementation of Strassens's matrix multiplication algorithm usually switch to the brute force method after matrix sizes become smaller than some crossover point. Run an experiment to determine such crossover point on your computer system.

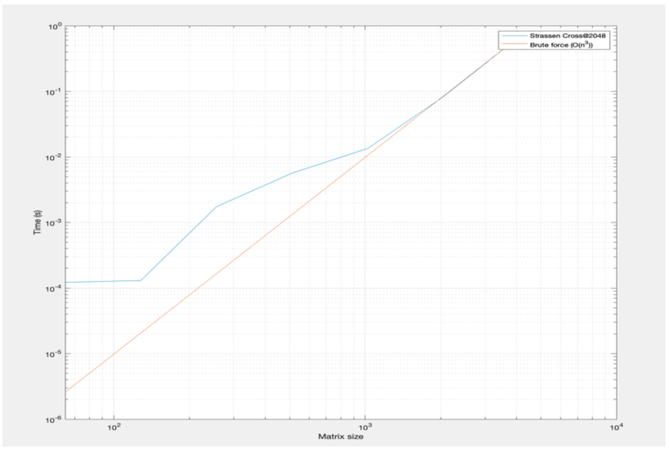
Code:

```
function [crossover_point, times] = strassen_crossover()

% matrix sizes
sizes = [2^6 2^7 2^8 2^9 2^10 2^11 2^12];

crossover_point = 2048.  % Crossover adjusted for different runs


% Initialize timing array
times = zeros(size(sizes));

for i = 1:length(sizes)

    % random matrices
    A = rand(sizes(i));
    B = rand(sizes(i));

    % Time Strassen's algorithm
    tic;
    C = strassen(A, B, crossover_point);
    times(i) = toc;

end
%------------------------------
figure;
loglog(sizes, times);
hold on;
loglog(sizes, times(end)*(sizes./sizes(end)).^3);
xlabel('Matrix size');
ylabel('Time (s)');
legend('Strassen Cross@2048', 'Brute force (O(n^3))');
grid on;

end
%-------------------------------
function C = strassen(A, B, crossover_point)

% Brute Force case
n = size(A, 1);
if n <= crossover_point
    C = A*B;
    return;
end

% Split matrices into submatrices
```

```
m = n/2;
A11 = A(1:m, 1:m);
A12 = A(1:m, m+1:n);
A21 = A(m+1:n, 1:m);
A22 = A(m+1:n, m+1:n);
B11 = B(1:m, 1:m);
B12 = B(1:m, m+1:n);
B21 = B(m+1:n, 1:m);
B22 = B(m+1:n, m+1:n);

% Strassen Algo
P1 = strassen(A11+A22, B11+B22, crossover_point);
P2 = strassen(A21+A22, B11, crossover_point);
P3 = strassen(A11, B12-B22, crossover_point);
P4 = strassen(A22, B21-B11, crossover_point);
P5 = strassen(A11+A12, B22, crossover_point);
P6 = strassen(A21-A11, B11+B12, crossover_point);
P7 = strassen(A12-A22, B21+B22, crossover_point);


C11 = P1+P4-P5+P7;
C12 = P3+P5;
C21 = P2+P4;
C22 = P1-P2+P3+P6;


C = [C11 C12; C21 C22];

end
```

Description: The program uses Matlab’s inbuilt timing function to test run times of the Strassen Multiplication algorithm. Typically the algorithm performs better on larger matrices and fails to be better than the brute force method of O(n3) complexity for small matrices. The ‘smallness’ is to be tailored according to the system and that is what we try to achieve here. Loglog graph is used to exaggerate differences in the performances of the two methods and also the graph for the brute force method is scaled and shifted to represent performance intuitively.
For the ARM 64 bit machine used for this simulation, the best case crossover point is deduced to be matrix of size 512.





122CS0067 Amiya Chowdhury

Q.12. 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*.

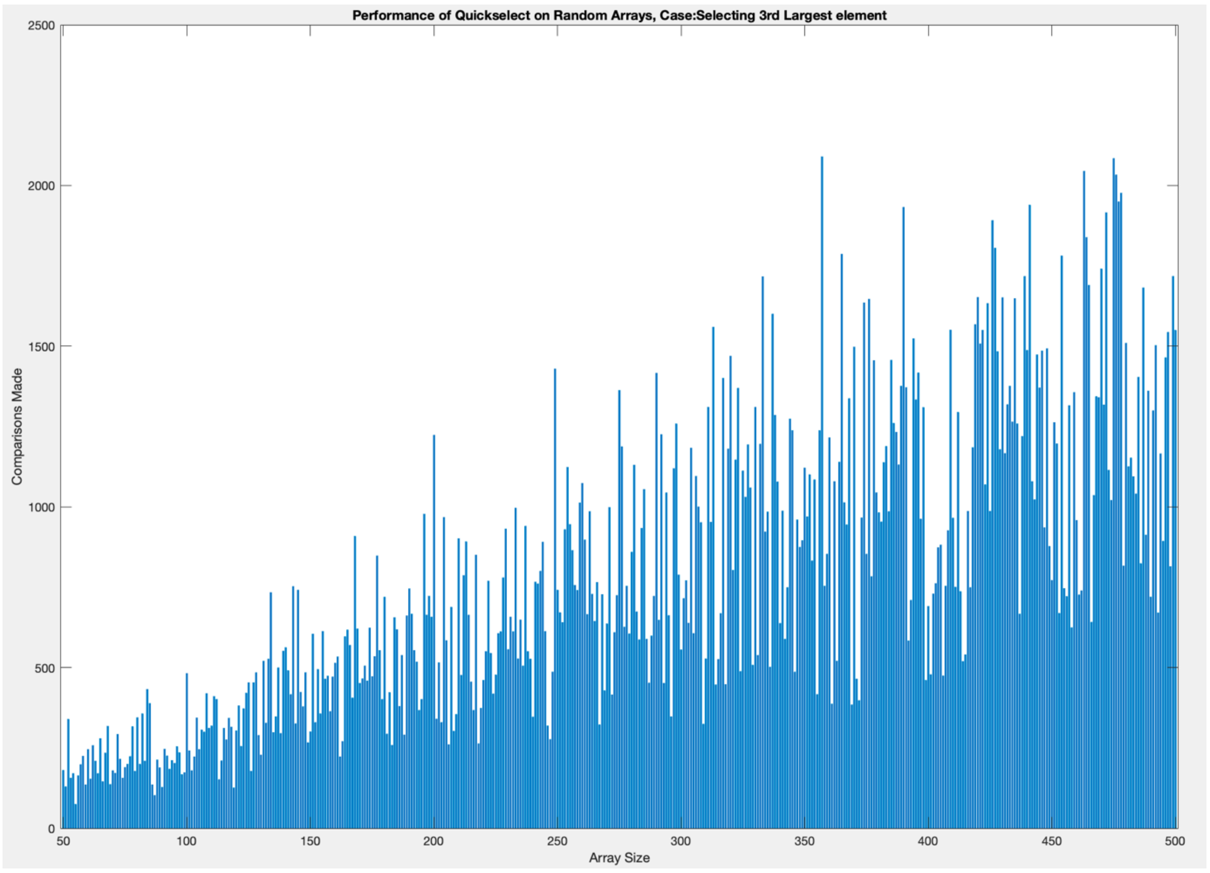
Code:

```
%-Quickselect Performance Analysis
%--In each instance 3rd largest element is found in arrays
%--of size ranging from 50-500
x=zeros(1,450);
y=zeros(1,450);
k=1;
for i=50:500
    arr=round(rand(1,i)*100);
    cp=quickselect(arr,1,i,0,3);
    x(k)=i;
    y(k)=cp;
    k=k+1;
end
bar(x,y);
title("Performance of Quickselect on Random Arrays, Case:Selecting
3rd Largest element");
xlabel("Array Size")
ylabel("Comparisons Made");

function cp= quickselect(a, lb, ub, cp,k)
if lb < ub
    cp = cp + 1;
    [a, loc, cp] = partition(a, lb, ub, cp);
    cp=cp+1;
    if(loc==k)
        res=a(k);
        return;
    elseif(loc>k)
        cp= quickselect(a, lb, loc-1, cp,k);
    else
        cp= quickselect(a, loc+1, ub, cp,k);
    end
end
cp = cp + 1;
end

function [a, d, cp] = partition(a, lb, ub, cp)
pivot = a(lb);
start = lb + 1;
last = ub;
while start <= last
    while start <= ub && a(start) <= pivot
        start = start + 1;
        cp = cp + 1;
    end
    cp = cp + 1;
```

122CS0067 Amiya Chowdhury

```
    while last >= lb + 1 && a(last) > pivot
        last = last -1;
        cp = cp + 1;
    end
    cp = cp + 1;
    if start < last
        temp = a(start);
        a(start) = a(last);
        a(last) = temp;
    end
end
a(lb) = a(last);
a(last) = pivot;
d = last;
end
```

Description: This program adheres to the problem statement given and represents the no. of comparisons or the cost required to select, in this case, the third largest element in an a randomly generated array. Observing, we find that it has the expected average asymptotic complexity of O(n) that is determined theoretically for the Quickselect algorithm.



122CS0067 Amiya Chowdhury

Q.14. Iterative Binary Search

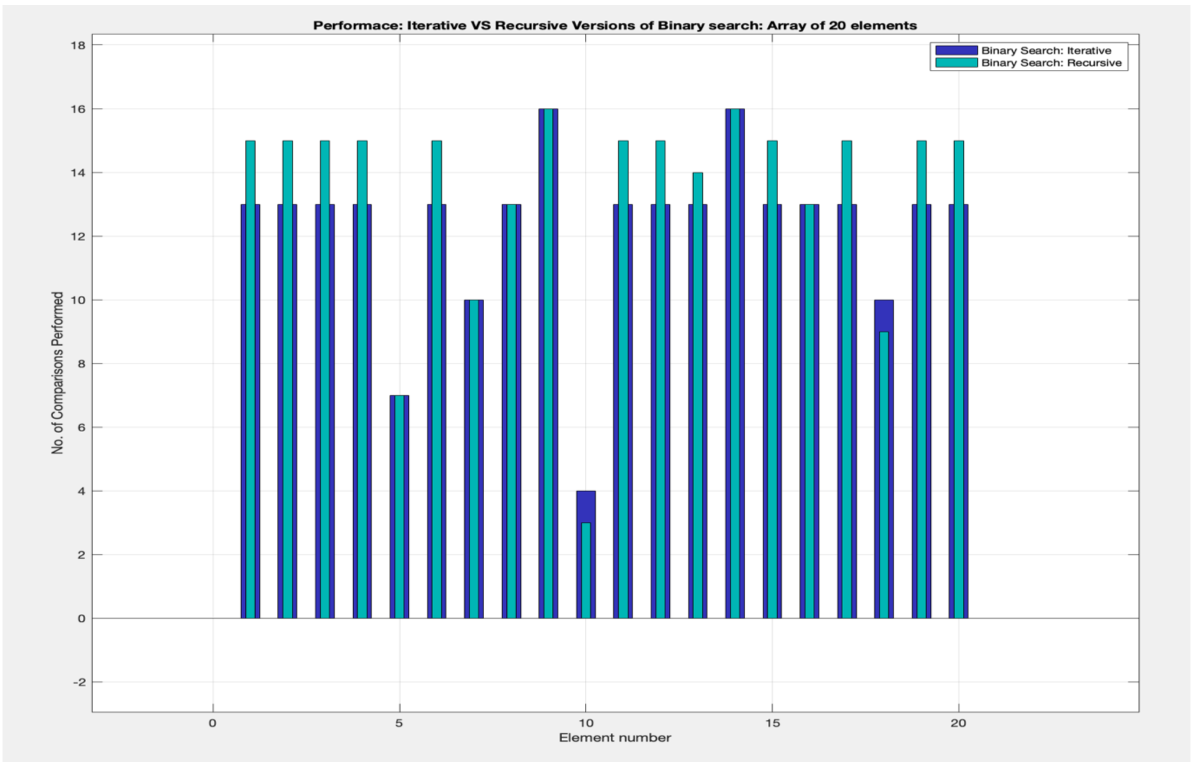
Write programs to implement recursive and iterative versions of binary search and compare the performance. For a appropriate size of n=20(say), have each algorithm find every element in the set. Then try all n+1 possible unsuccessful search. The performance, of these versions of binary search to be reported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

Code:

```
clear all;
sim_size=100;
step=10;
runs=10;
xne=zeros(1,20);
ync=zeros(1,20);
ync2=zeros(1,20);
arr=round(rand(1,20)*100);
for i=1:20
    xne(i)=i;
    ync(i)=bin_iter(arr,arr(i),1,20,0);
    ync2(i)=bin_recur(arr,arr(i),1,20,0);
end

bar(xne,ync,0.5,'FaceColor',[0.2 0.2 0.7]);
hold on;
bar(xne,ync2,.25,'FaceColor',[0 0.7 0.7]);
title("Performace: Iterative VS Recursive Versions of Binary search:
Array of 20 elements")
xlabel("Element number")
ylabel("No. of Comparisons Performed")
legend("Binary Search: Iterative","Binary Search: Recursive")
grid on

function cp = bin_iter(a,key,lb,ub,cp)
cp=cp+1;
while(lb<=ub)
    cp=cp+1;
    mid=floor((lb+ub)/2);
    cp=cp+1;
    if(a(mid)>key)
        ub=mid-1;
        cp=cp+1;
    elseif(a(mid)<key)
        lb=mid+1;
        cp=cp+1;
    else
        index=mid;
        cp=cp+1;
        return;
    end
end
end
```

122CS0067 Amiya Chowdhury

```
function cp = bin_recur(a,key,lb,ub,cp)
cp=cp+1;
if(lb<=ub)
    cp=cp+1;
    if(lb==ub)
        cp=cp+1;
        if(a(lb)==key)
            index=lb;
            return;
        end
    else
    mid=floor((lb+ub)/2);
    cp=cp+1;
    if(a(mid)==key)
        index=mid;
        return;
    elseif(a(mid)>key)
        cp=cp+1;
        cp=bin_recur(a,key,lb,mid-1,cp);
    else
        cp=bin_recur(a,key,mid+1,ub,cp);
    end
    end
else
index=-1;
end
end
```



122CS0067 Amiya Chowdhury

Description:
Both the iterative and recursive implementations of binary search tend to have the same performance. However the iterative version does seem to have a tiny bit of edge in this matter. We would come to the same conclusion if we tested execution times too as recursive implementations tend to have higher overhead cost due to a function call stack usage.