**1. Weiss 7.19**

3 1 4 1 5 9 2 6 5 3 5 // start

// medianOfThree(3,9,5) -> 5, swap pivot to centre and assign as pivot

3 1 4 1 5 5 2 6 5 3 9

^

p

3 1 4 1 5 3 2 6 5 5 9 // place pivot on the side

^

p

3 1 4 1 5 3 2 6 5 5 9 // set up left and right iterators

^ ^

i jp

3 1 4 1 5 3 2 6 5 5 9 // move left iterator until value >= pivot

^ ^

i jp

3 1 4 1 5 3 2 6 5 5 9 // move right iterator until value <= pivot

^ ^ ^

i j p

3 1 4 1 5 3 2 6 5 5 9 // swap

^ ^ ^

i j p

3 1 4 1 5 3 2 6 5 5 9 // move left iterator until value >= pivot

^ ^ ^

i j p

3 1 4 1 5 3 2 6 5 5 9 // move right iterator until value <= pivot

^ ^ ^

j i p

3 1 4 1 5 3 2 6 5 5 9 // no swap because iterators crossed

^ ^ ^

j i p

3 1 4 1 5 3 2 5 5 6 9 // swap pivot back into location with i

^

p

3 1 4 1 5 3 2 // divide and conquer. First half first

x

// medianOfThree(3,1,2) -> 2, swap and assign as pivot

1 1 4 2 5 3 3

^

p

1 1 4 3 5 2 3 // place pivot on the side

^

p

1 1 4 3 5 2 3 // set up left and right iterators

^ ^

i jp

1 1 4 3 5 2 3 // move left iterator until value >= pivot

^ ^

i jp

1 1 4 3 5 2 3 // move right iterator until value <= pivot

^ ^ ^

j i p

1 1 4 3 5 2 3 // no swap because iterators crossed

^ ^ ^

j i p

1 1 2 3 5 4 3 // swap pivot back into location with i

^

p

1 1 // divide and conquer, insertion sort

3 5 4 3 // divide and conquer

3 3 4 5 // medianOfThree(3,4,3) -> 3, swap and assign as pivot

^

p

3 4 3 5 // place pivot on the side

^

p

3 4 3 5 // set up left and right iterators

^ ^

i jp

3 4 3 5 // move left iterator until value >= pivot

^ ^

i jp

3 4 3 5 // move right iterator until value <= pivot

^ ^ ^

j i p

3 4 3 5 // no swap because iterators crossed

^ ^ ^

j i p

3 3 4 5 // swap pivot back into location with i

^

p

3 // divide and conquer, done

4 5 // divide and conquer, insertion sort

5 6 9 // divide and conquer, second half.

5 6 9 // medianOfThree(5,6,9) -> 6

^

p

5 9 6 // place pivot on the side

^ ^ ^

i j p

5 9 6 // move left iterator until value >= pivot

^ ^

ijp

5 9 6 // move right iterator until value <= pivot

^ ^ ^

j i p

5 9 6 // no swap because iterators crossed

^ ^ ^

j i p

5 6 9 // swap pivot back into location with i

^

p

5 // divide and conquer, done

9 // divide and conquer, done

1 1 2 3 3 4 5 5 5 6 9 // done

**2. Weiss 7.23**

This is pretty much asking whether the middle position of the array will

be the minimum or maximum (as this would cause quicksort to require

quadratic time). There are two possible answers, depending on justification.

One answer is that, assuming random distribution of data, choosing the middle

element will not help at all because the minimum and maximum are just as

likely to be in the middle spot as anywhere else.

The other answer is that, assuming most data is not completely random and

follows some level of sorting, choosing the middle element may be helpful

because minimum and maximum values will tend to be on the extremes rather than

the middle.

**3. Weiss 7.28a**

quicksort()

   array A of N elements

   partition(0, N, A)

partition(lo, hi, A)

   // base case

   if lo == (hi + 1), done

   leftPtr = lo

   leftEqual = lo

   rightPtr = hi

   rightEqual = hi

   Determine random pivot, p

   while(leftPtr != rightPtr)

       while(A[leftPtr++] <= p)

           if A[leftPtr] == p

               swap A[leftEqual], A[leftPtr]

               leftEqual++

       while(A[rightPtr--] >= p)

           if A[rightPtr] == p

               swap A[rightEqual], A[rightPtr]

               rightEqual++

       swap A[leftPtr], A[rightPtr]

   swap left at leftPtr from A[lo] to A[leftEqual]

   swap right at rightPtr from A[rightEqual] to A[hi]

   // Recursive calls to lower and upper

   partition(lo, leftEqual, A)

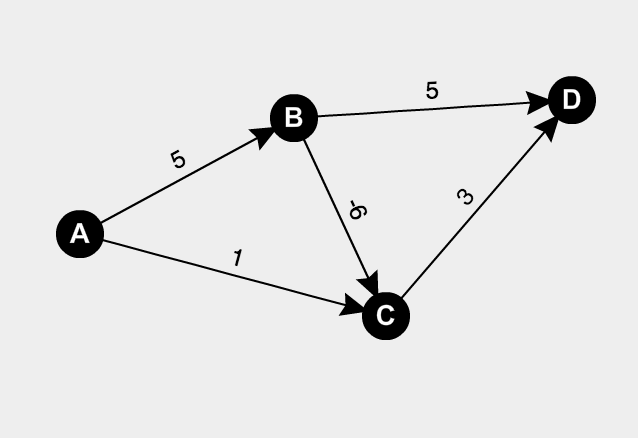
   partition(rightEqual, hi, A)

**4.Weiss 9.1**

Topological Ordering: S - G - D - A - B - H - E - I - F - C - T

There are several correct orderings outside of this ordering.

**5.Weiss 9.7a**



Dijkstra starting from A will explore vertices in the following order: C, D, B. In doing so, it will find A->C->D route to D, which has cost 4. It will miss A->B->C->D which has cost 2.

**6.Weiss Exercise 9.38**

You are given a set of N sticks, which are lying on top of each other in some con­figuration. Each stick is specified by its two endpoints; each endpoint is an ordered triple giving its x, y, and z coordinates; no stick is vertical. A stick may be picked up only if there is no stick on top of it.

a) Explain how to write a routine that takes two sticks a and b and reports whether a is above, below, or unrelated to b. (This has nothing to do with graph theory.)

Project all the sticks to the x­y plane. If the projections of two sticks do not intersect, we know they are unrelated; otherwise, suppose the intersection point of them is (p, q), then we can compute the z-­coordinate of each stick by substituting the x, y value of the line equation corresponding to each stick using p, and q respectively, and the stick with a larger z-­coordinate value is above the other.

Pseudo­code:

//returns the spatial relation between sticks a, b

SpatialRelationSticks(a,b){

If(projections of a and b to x‐y plane do not intersect)

//get x‐y slopes of sticks

//find point of intersection between sticks=(p,q)

//sticks intersect if point of intersection is within the bounds of both sticks

Return unrelated;

else{

compute the z‐coordinate of each stick (denoted as z(a)and z(b)) by

substituting the x, y value of the line equation corresponding to each

stick using p,and q respectively;

if(z(a)>z(b))

Return above;

Else

Return below;

}

}

b) Give an algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a sequence of stick pickups that accomplishes this.

Construct a directed graph G = (V, E) as follows. Each vertex of V corresponds to one stick. And a directed edge is constructed from vertex a to vertex b, if the corresponding sticks, denoted as s(a) and s(b), satisfy s(a) is above s(b). Then we can run Topological Sort algorithm on G. If there is cycle in G found by Topological Sort algorithm, we know that it is not possible to pick up all the sticks; otherwise, we follow the order of each stick produced by Topological Sort algorithm to pick them up.

Pseudo­code:

//returns the order of sticks picked up or if it wasn’t possible PickUpSticks(s(1),s(2),...,s(n)) //s(n)=listofnsticks

for(each stick s in list s(v))

Construct a vertex v for Graph G;

Project all sticks to x‐y plane;

Find all intersections between any two sticks by running the algorithm from(a)

between every pair of sticks;

if(spacial relation between two sticks)

add directed edge between vertices of two sticks to G;

G.topsort(); // topsort() method from Addison Wesley textbook

if cycle is found in G by topsort()

return No;

else

return the order of each stick produced by TopologicalSort;