Honors Data Structures
Theoretical homework 4

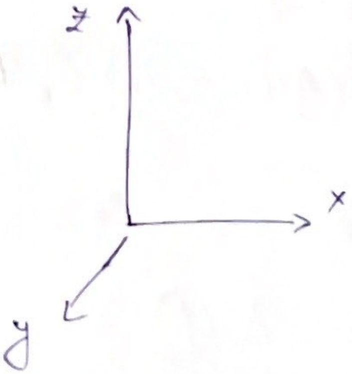
Mark Kirichev, mmk2243

Problem 1. Code is on the back of the document

Problem 2. The total running time to sort an array of length N using Merge Sort in this scenario is $O(N \log N)$. This happens because the time needed to recursively sort both partitions is $\max(T(N_1), T(N_2))$. Initially we might think that the complexity can be ~~[illegible]~~ $O(N)$ because of the parallel recursive calls. However, this would only be true if we could run every recursive call on ~~[illegible]~~ a new CPU core, meaning that essentially we would need an endless amount (or at least ~~[illegible]~~ a number, dependent on n) of CPU cores, which is obviously impossible => The runtime is $O(N \log N)$

Problem 3.

a) Let's choose a coordinate system such that the z-value is the "height value" of the stick.



Now, if some stick a is "above" or "below" some other stick b, then there will exist a pair (x, y) for which the point (x, y, z_a) is a point of a and the point (x, y, z_b) is a point of b.

This is true because if we ignore* the z-values of 3D points, we'll get their projection on the xy-plane and if the two sticks are not unrelated, then their projections onto xy-plane must cross somewhere. We can get the values of x and y through one of the following 2 ways:

1) Construct both sticks' orthogonal projections onto the xy-plane using techniques from linear algebra i.e a projection matrix, for example, which we multiply with the vectors formed by the parameterizations of the 3D line equations of the sticks. After getting the equations for both sticks' projections, we just solve the system of equations
$$\begin{cases} x_a = x_b \\ y_a = y_b \end{cases}$$

2) Solving the system of equations: $\begin{cases} f_a(t) = f_b(t) \\ g_a(t) = g_b(t) \end{cases}$

where the functions $f_a(t)$, $f_b(t)$, $g_a(t)$, and $g_b(t)$ are the parameterizing components of the sticks $\underline{\underline{a}}$ and $\underline{\underline{b}}$.

i.e. $\underline{\underline{a}} = (f_a(t), g_a(t), h_a(t))$; $\underline{\underline{b}} = (f_b(t), g_b(t), h_b(t))$.

The parameterizations can be acquired ~~by~~ from the sticks' ends' coordinates. For example, for $f_a(t)$:

$$f_a(t) = X_{a_{end\,1}} + t(X_{a_{end\,2}} - X_{a_{end\,1}})$$

for $t \in \mathbb{R}$
and $t \in [0; 1]$

After finding the xy-values of the intersection it's easy to find the z-values either through plugging the t_0 that we got into $h_a(t)$ and $h_b(t)$ since essentially we have $z_a = h_a(t_0)$; $z_b = h_b(t_0)$. Another way of finding the z-values is through the proportion:

$$\boxed{\frac{X_{final} - X_0}{X_0 - X_{initial}} = \frac{Z_{final} - Z}{Z_0 - Z_{initial}}}$$

After finding Z_a and Z_b we just have to compare them and we'll know which stick is on top. If the sticks do not have a common xy-pair (or their orthogonal projections onto xy-plane do not intersect), then the sticks are "unrelated".

Problem 3 continue:

This can occur if the system of equations does not have a ~~solutions~~ solution.

* ignore the z-values essentially means set them to 0.

6) The algorithm that we want is the following:

1) Define a graph G with n vertices.

2) Take every possible pair and apply the algorithm from part a). Since we have n vertices we'll have $\binom{n}{2} = \frac{n!}{2!\,(n-2)!} = \frac{n(n-1)}{2}$ pairs that we'd need to perform the algorithm on

3) We take every pair and for each pair:

3.1) If a is above b: we add an directed edge $a \to b$

3.2) If b is above a: we add an directed edge $b \to a$

3.3) If a and b are unrelated: we don't add edges

4) Search for cycles:
- 4.1) If there's a cycle, picking all sticks is impossible!
- 4.2) If there's no cycle, picking all sticks is possible!

5) If no cycles are evident: find topological order of the graph

6) That's the final order we can "untangle" the Mikado with.

Time	Visited	A		B		C		D		E		F		Priority Queue
		Cost	P	Cost	P	Cost	P	Cost	P	Cost	P	Cost	P	
0		0	-	inf	-	inf	-	inf	-	inf	-	inf	-	A
1	A	0	null	2	A	9	A	inf	-	inf	-	inf	-	B C
2	A B	0	null	2	A	8	B	5	B	10	B	inf	-	D C E
3	A B D	0	null	2	A	8	B	5	B	7	D	6	D	F C E
4	A B D F	0	null	2	A	7	F	5	B	7	D	6	D	C E
5	ABDFC	0	null	2	A	7	F	5	B	7	D	6	D	E
6	ABDFCE	0	null	2	A	7	F	5	B	7	D	6	D	empty

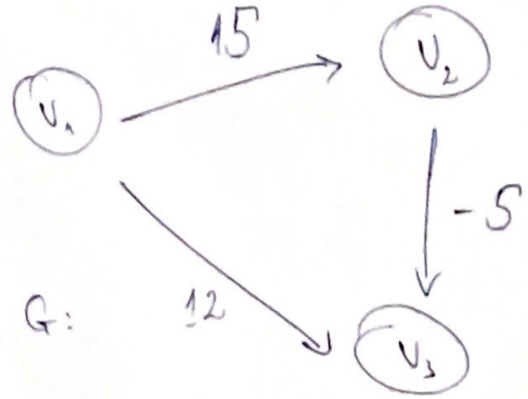
Problem 4.

Shortest path from A to E by following the backpointers:

E ← D ← B ← A

Problem 5.

Graph G:



The edges on the graph shown on the left are:

$V_1 \to V_2$, weight: 15

$V_1 \to V_3$, weight: 12

$V_2 \to V_3$, weight: -5

It is clear that we <u>don't</u> have any negative cost cycles in this construction. However, if we run Dijikstra's algorithm on it, trying to find the shortest path from v_1 to v_3, then we'd get the wrong result as Dijikstra would say that the solution is $v_1 \to v_3$ when the actual solution is $v_1 \to v_2 \to v_3$. (weight 12 vs weight 10). This mistake occurs because v_3 will be visited immetiately after v_1 is visited in the beginning. In general, negative weights introduce ~~the~~ a problem with the possibility of a significant decrease in total weight to a certain desired node later on and, thus, the path with less weight might be left unexamned. This is expected since Dijikstra's algorithm assumes that the general case would be "the more you traverse, the more you'd expect the path to weight" which is true for strictly positive edges.

```java
public static void threewaypartitionsort(Comparable[] array) {
    final int ARR_LENGTH = array.length - 1;
    threewaypartitionsort(array, 0, ARR_LENGTH);
}

threewaypartitionsort (Comparable[] array,
                       final int      low,
                       final int     high) {
    if (high <= low) {
        return;
    }

    int lesserThan = low;
    int greaterThan = high;

    Comparable x = array[low];
    int i = lesserThan;

    while (i <= greaterThan)
    {
        int data = array[i].compareTo(x);

        if (data < 0) {
            swapArrayValues(array, lesserThan++, i++);
        } else if (data > 0) {
            swapArrayValues(array, i,  greaterThan--);
        } else {
            ++i;
        }
    }

    threewaypartitionsort(array, low,   lesserThan - 1);
    threewaypartitionsort(array, greaterThan + 1, high);
```