XIV Algorithms ~~an~~ & Data structures

Assignment 14

I	True	False
1.	X	
2.	X	
3.	X	
4.		X
5.	X	
6.	X	
7.	X	
8.		X
9		X
10.		X
11.	X	

II input: adjecency list of undirected graph $G=(V,E)$
output: array of ~~all~~ twodegrees of all vertices.

Algorithm

1. Create an array of length $|V|$, called degree
2. For each $v \in V$, store the length of adjency row of v in the array. v_0 gets array[0], this way, array[0] contains the amount of neighbors v_0 has.
3. ~~for each~~ ~~To~~ Create a new array of length $|V|$ initialized to 0. ~~The~~ This array ~~is~~ is called twodegree.
4. For each edge, ~~the endpoints of edge e1 are~~ ~~called~~ and for both endpoints ~~add the~~ v_x and v_y. Add the degree of v_x to the twodegree of v_y, and the degree of v_y to the twodegree of v_x.

The complexity of this algorithm is:

1. $O(1)$
2. $O(|V|)$ (for each $v \in V$)
3. $O(1)$
4. $O(|E|)$ (for each $e \in E$)

Total $= O(|V| + |E|)$

III These bottleneck edges ~~are in the~~ all start in the min-cut of G. We ~~need to th~~ thus need to run Ford-Fulkerson to calculate the min cut:

1. Run Ford-Fulkerson
2. Find the reachable set of vertices (min-cut)
3. All edges from such a reachable ~~edge~~ vertex to a non-reachable vertex are bottleneck edges.

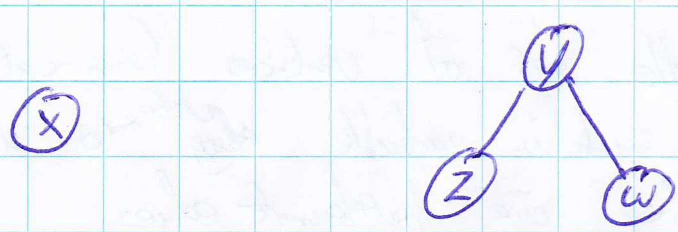
IV We can extend the binary search tree by keeping track of how many keys are smaller than a key:

For a key x, we keep track of:

x.key
x.smaller
x.parent
x.left
x.right

For insertion, there are two possibilities:

Inserting x in tree:



$x.key < y.key$: - y.smalle += 1
- inset x in z

$x.key \geq y.key$: - inset x in w.
- x.smalle += 1

For finding an node x for which there are k smaller keys, we can use this algorithm:

Algorithm: (using the same nodes as above)

1. If y.smaller == k, return y
2. If y.smalle < k, ~~search~~ ~~search~~ recursively use this algorithm to seach k - y.smalle ~~at~~ smaller keys in w. (y.right)
3. If y.smalle > k, recursively use this algorithm to seach k smalle keys in z (y.left)

This algorithm works because for a node w, there are w. ~~smaller~~ smaller keys in the tree plus the amount of smaller nodes the parent of w has. (because ~~it going rig~~ the right child of ~~the~~ a ~~key~~ node is always larger than the ~~the child of a~~ node itself).

V ~~For each item, we can calculate the value/weight ratio. This ratio will be used for adding items to the knapsack.~~

1. For a bottom-up approach, we use a 2D array K with n+1 columns and W+1 rows (n = amount of products).

Algorithm

```
KnapSackAlgorithm (int W, int weight[], int val[], int n) {

    int K[n+1][W+1]

    for (i=0; i<=n; i++) {
        for (we=0; we<=w; we++) {
            if (i==0 or we==0) {
                K[i][we] = 0;
            else if (weight[i-1]<= we) {
                K[i][we] = maximum(K[i-1][we],
                val[i-1] + K[i-1][we - weight[i-1]]);
            else K[i][w] = K[i-1][w]
```

either item ~~n~~ is in the knapsack or not, this way we can construct an algorithm:

$$K[i][w] = \begin{cases} 0 \text{ if } i == 0 \\ 0 \text{ if } w == 0 \\ \max(K[i-1][w],\ val[i-1] + K[i-1][w-weight[i-1]]) \\ \quad \text{if } weight[i-1] <= \text{remaining weight} \\ K[i-1][w] \text{ if } weight[i-1] > \text{remaining weight} \end{cases}$$

2. We can keep track of another bool array. This array indicates whether an item at place i is in the knapsack or not. This array will also keep track of the ~~previ~~ choice made:

```
struct Choise {
    ~~ch previous = choise | null~~ choise previous;
    bool in ~~true~~ | false
}
```

```
KnapSackAlgorithm(int w, int weight[], int val[], int n) {

   int K[n+1][w+1];
   choice C[n+1][w+1];

   for (i=0; i<=n; i++) {
       for (we=0; we<=w; we++) {
           if (i==0 or we==0) {
               K[i][we] = 0;
               choice C[i][we] = null;
           else if (weight[i-1] <= we) {
               K[i][w]
               if (val[i-1]+K[i-1][we-weight[i-1]] >=
                       K[i-1][we]) {
                   K[i][we] = val[i-1] + K[i-1][we-weight[i-1]];
                   C[i][we].in = true;
                   C[i][we].previous = C[i-1][we-weight[i-1]]
               else {
                   K[i][we] = K[i-1][we]
                   C[i][we].in = false;
                   C[i][we].previous = C[i-1][we]
               }
           else {
               K[i][we] = K[i-1][we]
               C[i][we].in = false
               C[i][we].previous = C[i-1][we]
           }
```

}

while

Choice c = C[k][w]

while (c != null) {

~~printC~~ if (c.in) {

~~print([c.c])~~ print(c);

}

}