מגישים : אושרי נורי (304984628)
יקיר חלץ (305028441)

2.1

1) **גרף** : נגדיר את ארונות החשמל ואת תחנות הכוח כצמתים ואת החיבורים בינם לבין עצמם כקשתות. שימוש במבנה נתונים זה יסייע לביצוע אבסטרקציה של רשת החשמל למבנה נתונים.

2) **רשימה מקושרת** : ראשית, נרצה לבטל את ההגבלה על מספר הפעולות - גודל מערך הינו חסום, ולכן כמות הפעולות האפשרית הינה מוגבלת. בנוסף, נרצה לשמור על יחס סדר בין הפעולות המבוצעות באופן כרונולוגי. לכן, רשימה עדיפה על סט. יתר על כן, ניתן לבצע את אותה הפעולה כמה פעמים. רשימה מקושרת מאפשרת לשמור את אותו איבר מספר פעמים (במקרה שלנו – פעולות) ולכן היא המבנה המתאים ביותר לכך.

3) **מילון** : ראשית, נרצה שלא יהיו ערכים כפולים במבנה שלנו ועל כן נפסול את השימוש ברשימה מקושרת/גרף. בנוסף, נרצה לאחסן את השמות (=מפתחות) בהתאמה חח"ע לפירושים שלהם (=ערכים). ניתן לבצע חיפוש בסט, אך הוא לא יהיה יעיל באותה מידה כמו מילון. ניתן לממש מילון בצורה השומרת על יחס סדר בין המפתחות, דבר המסייע לחיפוש בצורה יעילה יותר.

4) **רשימה מקושרת:** ראשית, נרצה לשמור על הסדר הכרונולוגי של ההזמנות – ולכן סט וגרף יורדים מן הפרק. אותו הלקוח יכול לבצע את ההזמנה מספר פעמים, ולכן נרצה לשמר את האפשרות לשמור כמה איברים דומים (הזמנות). רשימה מקושרת היא המתאימה ביותר.

2.2

```
typedef void* Element;
typedef bool (*CompareFunction)(Element a, Element b);

static void swap(Element* a, Element* b)    {
   void* temp = *a;
   *a = *b;
   *b = temp;
}

void quickSort(void** array, int size, CmpFunction compare){
   Element pivot;
   int b = 1;
   int t = size - 1;
   if (size < 2) {
      return;
   }
   swap(&array[0], &array[size/2]);
   pivot = array[0];
   while(b <= t)     {
      while(t >= b && compare(array[t],pivot) >= 0) {
         t--;
      }
      while(b <= t && compare(array[b],pivot) < 0) {
         b++;
      }
      if ( b < t)    {
         swap(&array[b++], &array[t--]);
      }
   }
   swap(&array[0], &array[t]);
   quickSort(array, t);
   quickSort(array + t + 1, size - t - 1);
}
```

```
void quickSort(void** array, int size, CmpFunction compare)
```

מערך של טיפוס כללי גודל המערך קריטריון השוואה

```
static void swap(void* a, void* b)
```

שני טיפוסים כללים בינהם יש לבצע החלפה

2.3 סעיף ראשון

```
static Node addNode(int n,Node nextNode) {
     Node node = malloc(sizeof(Node*));
     if (!node) {
          return NULL;
     }
     node->n = n;
     node->next = nextNode;
     return node;
}
static int getNum(Node node) {
     return node->n;
}
static void addRemainingNodes(Node* head, Node* joined) {
     while (*head != NULL) {
          (*joined)->next = addNode(getNum(*head),NULL);
          *joined = (*joined)->next;
          *head= (*head)->next;
     }
}
static void progressToNextNode(Node* head1, Node* head2) {
     if (getNum(*head1) <= getNum(*head2)) {
          *head1 = (*head1)->next;
     } else {
          *head2 = (*head2)->next;
     }
}
static int minValNode(Node node1, Node node2) {
     int a = getNum(node1);
     int b = getNum(node2);
     return (a<=b)?a:b;
}

Node joinSorted(Node head1, Node head2) {
     Node joined = NULL;
     Node joined_head = NULL;
     while (head1!=NULL && head2!=NULL) {
          Node newNode = addNode(minValNode(head1,head2),NULL);
          if (joined!=NULL) {
               joined->next = newNode;
               joined = newNode;
          } else {
               joined = newNode;
               joined_head = joined;
          }
          progressToNextNode(&head1,&head2);
     }
     addRemainingNodes((!head1?&head2:&head1),&joined);
     return joined_head;
```

2.3 סעיף שני

```
}
static Node copyList (Node node) {
    if (!node)     {
        return NULL;
    }
    Node copy = malloc(sizeof(*copy));
    copy->n = node->n;
    copy->next = copyList(node->next);
    return copy;
}

Node joinSortedArray (Node* array, int length){
     Node first = copyList(array[0]);
     Node joined_arr = NULL;
     for (int i=1 ; i<length ; i++)   {
          Node second = array[i];
          joined_arr = joinSorted(first, second);
          destroyList(first);
          first = joined_arr;
     }
     return joined_arr;
}
```