binary search tree that provides the most benefit and perform the copy. We will then recompute L and the binary search tree. We will continue to do those steps until we have run out of available space for redundancy. As a summary, the psuedo-code of this algorithm is shown as algorithm 1.

Start with the data units ~~which each with~~ at least one access requirement (AR);
Initialize ~~AR,~~ ~~heap~~ H, and ~~list~~ L;
for *each data unit* **do**
  Find number of ~~overlapping access requirements~~ and store the number with the data unit;
end
for *each AR P's head and tail data units U* **do**
  make old copy list L' empty;
  YOON's comment: unclear why you use this list. **if** *U is head data unit* **then**
    YOON's comment: no point to introduce S. just say copy data appropriate location. Let S = data unit in P after U;
  else
    Let S = data unit in P before U;
  end
  Set BENEFIT=distance(S,U);
  Let U' be the potential copy of U;
  Search data units between S and U for min number of overlapping ARs and put U' there;
  for *each AR T that also uses U* **do**
    if *T will be shorter by using U'* **then**
      Add T's benefit to BENEFIT;
    else
      Add T to list L';
    end
  end
  Add BENEFIT to heap H;
  if *L' is empty* **then**
    add U to list L;
  end
end
while ~~*BREAK has not been called*~~ **do**
  while *L is not empty* **do**
    ~~Take out random element and move the data unit;~~
    ~~Update heap H and list L;~~
  end
  if ~~*there is more space for redundancy*~~ **then**
    ~~pop best element U from H;~~
    ~~copy~~ U to its destination;
    ~~update~~ nodes in H and entries in L for affected access requirements;
    update H and L;
  else
    ~~call BREAK;~~
  end
end

Algorithm 1: Pseudo-code for our algorithm

4 ~~Run-time and Storage~~ Analysis

We now analyze the running time and storage requirements of our algorithm. ~~We will denote~~ N ~~as~~ the number of data units and A ~~as~~ the number of access requirements. We will use k as the average span of a single access requirement. ~~The variable Q will represent how many executions of the redundancy loop occur.~~ The average number of overlapping access requirements is proportional to the number of data units multiplied by the redundancy factor r, which is the amount of redundancy if we have a single-seek layout. Therefore, the number of overlapping access requirements is $O(rN)$.

4.1 ~~Run-time Analysis~~

The ~~loop that initially constructs the access requirement heap is run on all A access requirements. It~~ involves ~~finding all~~ the benefit information ~~and putting it into a heap so that it is easy to figure out the best access requirement to modify first.~~ ~~With each run of the loop, in addition to a constant number of initial operations, there are $log(A)$ operations to insert the data into the heap plus $O(k)$ operations for the search of the AR to get the benefit information.~~ Thus it takes a total of $O(A(k + logA))$ operations to do the initial construction.

~~With the following loop which actually copies the data units and updates the information, there are a total of Q executions of it. In addition to the constant number of operations, each execution of the loop has to go through each of the overlapping access requirements and update their data. There are $O(rN)$ overlapping access requirements. For each of the affected access requirements, there are $O(k + log(A))$ operations to recalculate the benefit data and reform the heap.~~ Thus the final loop takes $O(QrN(k+logA))$ operations.

This means that in total, our algorithm takes $O((QrN + A)(k + logA))$ operations. In all likelihood, we will have to run the loop at least once, so $Q \geq 1$. Additionally, since $r \geq 1$, we know that $rN \geq A$. Thus we can simplify the expression to say that our algorithm takes $O(QrN(k + logA))$ operations. ~~Because we only have polynomial or logarithmic terms, we have found an algorithm that is efficient given our input size for computing an optimized layout.~~

4.2 ~~Storage Analysis~~

During the run of the algorithm, we have to store the number of overlapping access requirements at each data unit, which will require $O(N)$ storage. We will also have to store a heap of access requirements, which can be stored using $O(A)$ space. We also have a list of access requirements and that information will take up $O(A)$ space. In total we thus have $O(A + N)$ storage space used during the run of the algorithm.

4.3 ~~Linear search justification~~

~~Here I justify why we do a simple element by element search on the set of possible data units in the part of the algorithm where we decide where to put the copy. The original problem is to find the least number in an arbitrary block of a list. This is because each data unit stores the number of overlapping access requirements so we have a list of numbers to search through. There are only a limited number of data units where we could move the copy to hence we only care about an arbitrary part of this list.~~ If k is the ~~size~~ of the access requirement, then the ~~searching gives us~~ $O(k)$ query time. Updates will also be $O(k)$ and ~~construction will be $O(N)$ with N being the number of data units.~~ There are other approaches, such as a range tree or dynamic programming, that may produce better query times, but their construction and update times will be worse as well as their storage.

With dynamic programming, we would have to maintain a matrix where an entry (i, j) would contain the minimum value in that range. This would give us a $O(1)$ query time but the construction and storage would be $O(N^2)$ where N is the number of data units. The update time would be $O(N)$ when we add a data