University of Toronto at Scarborough
CSCC73 - Algorithm Design and Analysis

Assignment #3: Divide and Conquer

1. (10 marks) You have been given a challenge by your younger sibling to sort their bucket of magnets. (S)he has a bucket of n magnets that appear to be identical however some are positively charged and some are negatively charged. There are more positively charged magnets than negatively charged. When you put two oppositely charged magnets together they attract each other. When two identically charged magnets are put together, they repel each other. Give a divide and conquer $\mathcal{O}(n)$ algorithm to sort the magnets into positives and negatives.

   Solution A.

   Divide and conquer solution. Line the magnets up. Split in half and recursively split each half into two groups A1 and B1 and A2 and B2. Store groups as a doubly linked list with a counter list length. Take one magnet from A1 and one from A2 and compare. IF they are are the same, append A2's list to A1s and B2's to B1. If they are different, append B2 to A1 and B1 to A2 and update the list lengths.

   With the final two lists, compare counters. Larger counter is positive. The complexity can be given by the recurrence by T(n) = 2T(n/2) + O(1) which by the master theorem is O(n).

```
FindPositive(D[1 .. n]):
# Given an array of magnets, return two groups or like type magnets and their sizes.
    list A, list B1;
    sizeA = 0;
    sizeB = 0;
    if n == 1
        return (A1.append(D[1]), B1, 1, 0)

    (A1, B1, sizeA1, sizeB1) = FindPositive(D[1..n/2]);
    (A2, B2, sizeA2, sizeB2) = FindPositive(D[n/2+1 .. n]);
    # merge lists
    if A1 and A2 repel:
        # same type
        A1.append(A2);
        sizeA = sizeA1 + sizeA2;
        B1.append(B2);
        sizeB = sizeB1 + sizeB2;
    else
        A1.append(B2);
        B1.append(A2);
        sizeA = sizeA1 + sizeB2;
        sizeB = sizeB1 + sizeA2;
    return(A1, B1, sizeA, sizeB);

# Call to find groups and label groups
(A, B, sizeA, sizeB) = FindPositive(D[1..n])
if sizeA > sizeB:
    label A as positive;
else label B as positive
```

Correctness is simply given by induction. The base case with one magnet obviously returns the single group. Given that $A1, A2, B1, B2$ and their sizes are correctly returned by the induction hypothesis we just confirm that the merge works correctly. Note that we do not know which of $A1, A2, B1, B2$ are positive and negative but that $A1$ and $B1$ are different and $A2$ and $B2$ are different. Therefore, $A1$ must either be the same as $A2$ (repel) or different and therefore the same as $B2$. Similarly for $B1$. Appending the same types (repel) for As means that both B's must be the same. Conversely, if $A1$ is not the same type as $A2$ it must be the same type as $B2$ (and therefore $B1$ is the same type as $A2$). Since we know that there are more positive magnets than negative magnets we can easily identify the larger group as positive,

Solution B.

Solution (divide and conquer): I'll describe a recursive algorithm to identify one positively charged magnet. After this algorithm runs, we can identify all positively and negatively charged magnets in O(n) additional time, by comparing the chosen magnet to all the others.

(a) Pair up the magnets, and touch each pair to each other.
   i. If a pair are attracted to each other, toss them out
   ii. If a pair repel each other, put one of the pair into the bucket and throw the other one out

(b) If n is odd, add the last magnet to the bucket.

(c) Recursively find a positive magnet in the bucket.

```
OnePositive(D[1 .. n]):
  if n == 1
    return D[1]
  m = 0
  for i = 1 to ⌊n/2⌋
    if SameCharge(D[2i - 1], D[2i])
      m = m + 1
      Bucket[m] = D[2i]
  if n is odd
    m = m + 1
    Bucket[m] = D[n]
  return OnePositive(Bucket[1 .. m])
```

Correctness.

We prove this algorithm correct by induction, paying careful attention to the case where n is odd. There are two cases to consider.

(a) The algorithm is trivially correct when n = 1.

(b) So suppose $n > 1$. Again, let m denote the number of positive magnets, and let ℓ be the number of positive magnets put into the bucket. Then exactly $m - \ell$ positive magnets are not put into the bucket; of those, at most ℓ were paired with other positive magnets (When n is odd and $D[n]$ is a positive magnet, only $\ell - 1$ positive magnets are paired with other positive magnets.). So at least $m - 2\ell$ positive magnets were paired with negative magnets. Thus, at least $m - 2\ell$ negative magnets were paired with positive magnets, leaving at most $(n - m) - (m - 2\ell) = n - 2m + 2\ell$ remaining negative magnets, therefore at most $n/2 - m + \ell$ negative magnets are in the bucket. Since $m > n/2$, the number of negative magnets in the bucket is strictly less than ℓ. So the induction hypothesis implies that the recursive call correctly returns a positive magnet.

Complexity The running time obeys the recurrence $T(n) \leq O(n) + T(\lceil n/2 \rceil)$, because at most half the magnets are put in the bucket. Ignoring the ceiling in the recursive argument, we get a descending geometric series, so overall the algorithm runs in $O(n)$ time. In fact, the number of comparisons is exactly $n - \#1(n)$, where $\#1(n)$ is the number of 1s in the binary representation of n. This is the best possible.

2. (10 marks) In lecture we looked at two sequences of numbers and measured their similarity by counting the number of inversions. As in lecture we will consider a sequence of n numbers $x_1, x_2, \ldots, x_n$ which we assume are all distinct. We defined an inversion to be a pair $i < j$ such that $a_i > a_j$. In this situation, we make the definition of an inversion coarser and now call a pair an inversion if $a_i > 2a_j$. Give an $\mathcal{O}(nlogn)$ algorithm to count the number of coarse inversions.

Solution.

We'll define a recursive divide-and-conquer algorithm which takes a sequence of distinct numbers $a_1, a_2, \ldots, a_n$ and returns N and $a'_1, a'_2, \ldots, a'_n$ where

- N is the number of significant inversions
- $a'_1, a'_2, \ldots, a'_n$ is the same sequence sorted into increasing order

The algorithm is similar ot hte one from class that computes the number of inversions. The difference lies in the 'conquer' step where we will now merge twice: first we merge $b_1, \ldots, b_k$ with $b_{k+1}, \ldots, b_n$ just for sorting and then we merge $b_1, \ldots, b_k$ with $2b_{k+1}, \ldots, 2b_n$ for counting significant inversions.

Let's formally define the algorithm. For $n = 1$, $N = 0$ is returned and $\{a_1\}$ for the sequence. For $n > 1$ the algorithm does the following:

- Let $k = \lfloor n/2 \rfloor$.
- call the algorithm on $(a'_1, \ldots, a'_k)$. Say it returns N_1 and $b_1, \ldots, b_k$.
- call the algorithm on $(a'_{k+1}, \ldots, a'_n)$. Say it returns N_2 and $b_{k+1}, \ldots, b_n$.
- compute the number N_3 of significant inversions (a_i, a_j) where $i \leq k < j$.
- return $N = N_1 + N_2 + N_3$ and $a'_1, a'_2, \ldots, a'_n = \text{MERGE}(b_1, \ldots, b_k; b_{k+1}, \ldots, b_n)$.

MERGE can be implemented in $\mathcal{O}(n)$ time. All that remains is to find a way to compute N_3 in $\mathcal{O}(n)$ time as the there are $\mathcal{O}(\log n)$ recursive calls (as seen in class). We'll implement a variant of the merge count seen in class:

- Initialize counters: $i \leftarrow k$, $j \leftarrow n$, $N_3 \leftarrow 0$.
- If $b_i \leq 2b_j$ then
  - if $j > k + 1$ decrease j by 1.
  - if $j = k + 1$ (we are done) return N_3.
- If $b_i > 2b_j$ then
  - increase N_3 by $j - k$.
  - if $i > 1$ decrease i by 1.
  - if $i = 1$ return N_3.

Explanation For every i we count the number of significant inversions between b_i and all b_j's. If $b_i \leq 2b_j$ then there are no significant inversions between b_i and any b_m s.t. $m \geq j$, so we decrease j.

If $b_i > 2b_j$ then $b_i > 1b_m$ for all m s.t. $k < m \leq j$. In other words we have detected $j - k$ significant inversions involving b_i. We increase N_3 by $j - k$. Finally when we are down to $i = 1$ and have counted significant inversions involving b_1. there are no more significant inversions to be detected. This "*walk*" through $b_1, \ldots, b_k$ and $b_{k+1}, \ldots, b_n$ take $\mathcal{O}(n)$ time for an $\mathcal{O}(n \log n)$ time.