Lecture 23, Dec 4, 2023

Analyzing Problem Complexity – Adversary Approach

- Example problem: Find both the minimum and maximum of a set S of n distinct integers
- Naive algorithm: scan S twice, first to find the maximum, and then find the minimum
  - Finding the max takes $n-1$ comparisons exactly
  - Finding the minimum then takes $n-2$ (since we don't have to compare against the max we just found)
  - Therefore total is $2n-3$ comparisons
- Improved algorithm: divide S into $\frac{n}{2}$ pairs and find the maximum and minimum of each pair; then scan all the maxes to find the max, and all the mins to find the min
  - Initially $\frac{n}{2}$ comparisons to find the max and min of each pair, then $\frac{n}{2}-1$ comparisons each to find the max and min
  - Therefore the total is $\frac{3n}{2}-2$ comparisons
- Another algorithm is a divide-and-conquer approach of first dividing the set into 2, finding the min and max of each set, and then compare those
  - This gives the same number of comparisons as the above algorithm, however
- Theorem: Any comparison-based algorithm to solve this problem makes at least $\frac{3n}{2}-2$ comparisons in the worst case
- We prove this by using an *adversary argument*: given any algorithm, the adversary will come up with an input that forces the algorithm to do at least $\frac{3n}{2}-2$ comparisons
- At any point, we can categorize every element in the input set into 4 subsets: N – never compared, W – won every comparison so far, L – lost every comparison so far, M – won some and lost some comparisons
  - Initially, the size of N is n, while every other set has size 0
  - When the algorithm finishes, the size of N will be 0, the size of W and L are both exactly 1, and the size of m is $n-2$
- Intuitively, the maximum will win all comparisons, and the minimum will lose all comparisons; all other nodes have mixed comparison results
  - The adversary wants to delay the creation of mixed comparison results as much as possible
  - Note the adversary must not create cycles as to keep the input valid
  - The rough idea is we want elements in W to keep winning comparisons, and elements in L to keep losing, to delay populating M for as long as possible
- Adversary's strategy:
  - Compare N to N: assign arbitrarily, increasing W by 1 and L by 1
  - Compare N to W: N loses, increasing L by 1 and keeping W the same
  - Compare N to L: N wins, increasing W by 1 and keeping L the same
  - Compare N to M: N wins, increasing W by 1 and keeping M the same
  - Compare W to W: one wins, increasing M by 1 and decreasing W by 1
  - Compare W to L: W wins, keeping both the same
  - Compare W to M: W wins, keeping both the same
  - Compare L to L: one wins, increasing M by 1 and decreasing L by 1
  - Compare L to M: M wins, keeping both the same
  - Compare M to M: assign arbitrarily, keeping both the same
- Claim: by following this strategy, we can always produce inputs that are consistent (i.e. no cycles) and forces the algorithm to take at least $\frac{3n}{2}-2$
  - Starting from n elements all in N, any algorithm must:
    1. Create $n-2$ elements in M
       * This only happens when we compare W to W or L to L
       * We need exactly $n-2$ comparisons of this type to create the elements we need in M

2. Create n elements in W or L ($n-2$ of which will be changed to M, with the last 2 remaining)
   * The best way to do this is by comparing N to N, which creates 1 of each W and L
   * Therefore we need $\frac{n}{2}$ comparisons to create the n that we need

– Therefore the algorithm must perform at least $n-2+\frac{n}{2}=\frac{3n}{2}-2$ comparisons to reach the result