Samuel Sicklick

Algorithm

This implementation of the *MaximizePayout* sorts the inputted lists and then applies the payout of $\prod_{i=0}^{n} a_i^{b_i}$, as defined in the requirements, for each index i in both (now sorted) lists. This algorithm is greedy by its decisions being based on some fixed and simple "priority" rule,[1] which is the intuitive approach to raise the highest values of each first. This method should produce the maximum payout due to the weight of higher bases raised to higher powers, which will exponentially overshadow the later processed lower payouts.

Proof of Correctness (Optimality)

1. There is some value produced by any algorithm (ordering of inputs) since the data from the lists is just plugged into the payout formula described above. (See *Algorithm*)
2. Let O be some optimal algorithm (ordering of inputs) which produces the maximum payout, and G be the greedy algorithm described above. (See *Algorithm*)
    a. Counter Example to pairing Smallest Bases to Highest Exponents:
        i. If $A = [1, 2, 3], B = [3, 2, 2]$ then the payout would be 36, while the greedy algorithm described above would produce 108.
    b. Counter Example to pairing Highest Bases to Smallest Exponents:
        i. If $A = [1, 2, 3], B = [3, 2, 2]$ then the payout would be 36, while the greedy algorithm described above would produce 108.
    c. Counter Example to Unsorted Pairing:
        i. If $A = [1, 2, 3], B = [3, 2, 2]$ then the payout would be 36, while the greedy algorithm described above would produce 108.
3. O_i will produce a value greater than or equal to that of G_i:
    a. If G_i were to produce a value greater than O_i then O would have failed to be optimal.
4. G_i will produce a value greater than or equal to that of O_i:
    a. If O_i were greater than G_i then there must have been some base and exponent combination in G which did not increase the payout as much as the corresponding combination did in O.
    b. The most drastic increases will be caused by combinations of the highest base and highest exponents.
    c. Thus, G must have had some i where the corresponding values in A and B were not proportionally as large as the combination produced by O.
    d. By contradiction, since G will always pair its highest bases and exponents together, this cannot be.
5. Thus, G must be optimal, since it cannot produce a lower payout than any other solution.
6. QED

[1] Slide deck "Intro. To Greedy Algorithms" slide 6.

Samuel Sicklick

Running Time

1. Sorting each list costs $nlogn$ time, as it is the java.Collections.sort which implements mergesort.[2]
    a. This totals $2nlogn$.
2. Iterating through the lists costs n, as it traverses all of the inputs given, and a constant cost of multiplying the product of earlier steps with the result of raising the base to the given exponent in the next iteration.
    a. This independently costs n, raising the total time to $3nlogn$.
3. Thus, the algorithm runs in $O(nlogn)$.
4. QED

[2] See JavaDocs that this implementation actually is faster than regular mergesort, but we'll still with the given measure for simplification.