1. Japanese Fagaku Supercomputer instructions per second. Algorithm A needs instructions to finish for any input size of . How many Centuries would it take to finish the A.

Seconds to completion

Centuries to completion

1005 Centuries.

Seconds to completion

Centuries to completion

8.5 Nonagintillion Centuries.

1. Order the list of functions in asymptotically increasing order

Smallest

Largest

1. For each pair indicate the case: , , .
2. )
3. )
4. )

4. Knapsack problem

1. Algorithm Description

Using two loops my algorithm finds the “two approximate solution for the knapsack problem”. Checking every combination of numbers in the items list, my algorithm finds a solution in time.

It compares the first item in the items list against K / 2 and K. If that number satisfies the condition then it returns that number.

A running total for this loop is also kept.

Then it grabs the next item in the items list and compares it against the stopping condition.

If the next item in the list makes the total go higher than K then that item is disregarded from the total and the next item in the list is checked.

If the total at any point meets the stop condition then the list of numbers that made the total is returned.

1. Pseudocode

give K and items list

loop item1 in items

if item1 > K then start the loop again with next item

if item1 >= K / 2 then return item1

solution is a new list

total starts at 0

solution add item1

total add item1

loop item2 (item1+1 index) in items

if item2 > K then start the loop again with next item

if item2 >= K / 2 then return item2

solution add item2

total add item2

if total > K then total sub item2 and solution remove item2

if total >= K /2 return then solution

return null

1. Correctness

Every combination of numbers is looked at in the items list. The only time my algorithm would return a solution is if a number or group of numbers added to K / 2 -> K inclusive. If it doesn’t find a number or a combination of numbers that fit the stop condition then a null value is returned. It would only ever return null if there is no combination of items to meet the stopping condition in the items list. I know it checks every pair of numbers together because it keeps checking for numbers that could satisfy the stopping condition. If a number brings the running total to high then that number is discarded and the total is continued. This way every and all combinations of numbers can be compared.

1. Time Analysis

Every number is checked with every other number making it have comparisons.. The overall structure of my algorithm consists of a pair of nested loops. Which would mean that it has a time complexity of . This also makes sense given the number of comparisons made.

If I count the loops and comparisons done on the algorithm I would get this function:

.

Throwing out constants I would have:

Because would grow much faster than , the time of the algorithm would best be described using since that is the real growth of the algorithm's time complexity.

Thus:

Here is my java code if you would like to see it but it's not necessary to read it.

public static double[] solve(double K, double[] items) {

for (double item1 : items) {

if (item1 > K) continue;

if (item1 >= K / 2) return new double[]{item1};

List<Double> solution = new ArrayList<>();

double total = 0;

solution.add(item1);

total += item1;

for (int j = 1; j < items.length; j++) {

double item2 = items[j];

if (item2 > K) continue;

if (item2 >= K / 2) return new double[]{item2};

solution.add(item2);

total += item2;

if (total > K) {

total -= item2;

solution.remove(item2);

}

if (total >= K / 2) return solution.stream().mapToDouble(Double::doubleValue).toArray();

}

}

return null;

}