A computational problem

Dr. Hassanain Al-Taiy

Consider a list of positive integers. We are given a positive integer k and wish to find two (not necessarily distinct) numbers, m and n, in the list *whose product is* k, i.e. $m \times n = k$.

For example, $k = 72$, and the list is

$$[5, 24, 9, 5, 30, 6, 3, 12, 2, 10].$$

How do we do this in general?

We need an algorithm for this computational task, that is we need to describe a step-by-step process which we can implement as a program.

DO IT! HOW MANY ALGORITHMS CAN YOU SUGGEST?

Approaches to developing algorithms

There are many algorithmic techniques available.

For this problem, here are some possibilities:

- We may search the list directly.
- We may try to preprocess the list and then search.
- We may try to use the product structure of integers to make a more effective search.
- Others?....

Let us try the simplest possible exhaustive search.

In pseudocode , using an **array** A of positive integers, we might write this as:

```
product-search(int A[])
  found <- false;
  { for (i from 0 upto length(A))
      { for (j from 0 upto length(A))
              { if ( A[i]*A[j] = k )
                  then { found <- true, return; }
               } }
  }; return;
```

We could (usefully) return the found values!

Is this a good algorithm? How do we compare algorithms? What is a useful measure of the performance of an algorithm?

An algorithm using preprocessing

Consider an algorithm in which we first sort the array into (say) ascending order.

For example, the result of the sorting may be

$$[2, 3, 5, 5, 6, 9, 10, 12, 24, 30].$$

Can we search this list 'faster'?

Searching a sorted list: idea

Dr. Hassanain Al-Taiy

Idea: search from both ends! Why? Let us see what happens...

If the product is too small, increment left position; if too large, decrement right position.

Let us try it on our example, to find two numbers with product 72 in the list

$$[2, 3, 5, 5, 6, 9, 10, 12, 24, 30].$$

Start with 2 and 30. Then $2 \times 30 = 60 < 72$ so move left position along one and try $3 \times 30 = 90 > 72$, so move right position down the list one and try $3 \times 24 = 72$, BINGO!

It worked on this list, but can we show it works for every list. That is we need a correctness argument.

Searching a sorted list - algorithm

This gives an algorithm. In pseudocode for arrays in ascending order:

```
product-search(int A[])
  found = false;
  i <- 0; j <- length(A);
  while (i =< j)
    { if ( A[i]*A[j] = k )
           then { found <- true; return }
           else if ( A[i]*A[j] < k )
                  then { i <- i+1 }
                  else { j <- j-1 }

     };
  return;
```

Is this algorithm (a) correct, and (b) any better than the first?

Lect. 6 25/03/24

Dr. Hassanain Al-Taiy

A correctness argument

We need to show that the above algorithm does not overlook any candidate pair of numbers.

Consider an array A of integers in ascending order, and suppose the left position is i and the right position is j.

Suppose $A[i] \times A[j] < k$.

Then either j is the final position in A, in which case we must increment i; or we reached j by decrementing. In this case, there is a position x such that $0 \leq x \leq i$ and $A[x] \times A[j+1] > k$. Then, $A[i] \times A[j+1] > k$, so no elements to the right of j can contribute to a candidate pair. Likewise, no elements to the left of $i+1$ can contribute to a candidate pair, so we increment position i to $i+1$.

Likewise for the other case: $A[i] \times A[j] > k$.

Lect. 6 25/03/24

Dr. Hassanain Al-Taiy

Measures of performance and comparing algorithms in practice

What do we measure?

We count the number of operations required to compute a result.

Which operations?

- Operations should be significant in the running time of the implementation of the algorithm.
- Operations should be of constant time.

This is called the time complexity of the algorithm, and depends on the input provided.

Time complexity of the naive searching algorithm

How many operations does the first algorithm take?

Which operations? Either multiplication or equality (it doesn't matter).

Suppose the input is an array of length N.

Best case: It could find a result with the first pair, in which case we need just 1 operation.

Worst case: It could find the result as the last pair considered, or not find a result. Need N^2 operations (1 for each pair).

Lect. 6 25/03/24
Dr. Hassanain Al-Taiy

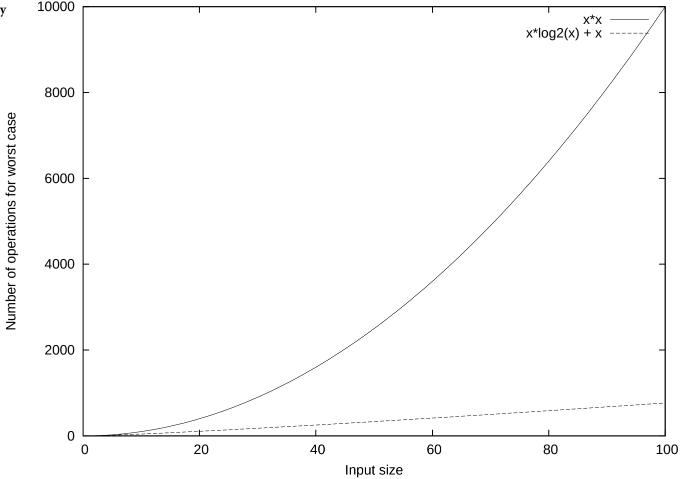
For second algorithm: Number of operations $=$ Number required for sorting $+$ number required for searching.

For sorting we can do this quite fast: For array length N, we can sort it in approx. $N \times \log_2(N)$ comparison operations (see later).

Note: $\log_2(N)$ is much smaller that N for most N, so $N \times \log_2(N)$ is much smaller than N^2.

How many operations for the searching? Answer: best case is 1 (again) and worst case is N (each operation disposes of one item in the array).

What about the average case? For these algorithms, the worst case is a good measure of the average case - but not always. So this algorithm is much better than the naive search using these measures.



Plot: Upper curve is the naive search, lower is the algorithm using sorting.