

Due: Thursday, August 9th, 6PM

Worth: 4%.

submissions: Use CDF submit command. Assignment name is PA2.

File name: PA2sol.c, PA2sol.cc, PA2sol.java or PA2sol.py

Test Cases: The test cases were generated as follows:

- The first 4 are either from the examples in the assignment handout or manually designed.
- The next 11 are square boards of increasing size up to the maximum, with randomly generated entries, to test the scalability.
- The last 5 are completely randomly generated, not necessarily square.

If your program did not get some of the test cases 10 to 15 it is probably because it was too slow.

The 0/20 marks: We checked all the 0/20 to make sure there was no problem with the automarker. Here are some of the reasons people got 0/20:

- Java submission does not compile/run with the commands we provide on the webpage.
- Incorrect formatting; as we mentioned we fixed these by hand for Programming Assignment 1 but are not going to do the same for Programming Assignment 2.
- Plain wrong solutions. Your program must produce wrong answer for the examples provided in the assignment statement for you to get 0.

The runtime: The PA2sol.cc and PA2sol.java files on the course webpage solve question using a simple dynamic programming solution that runs in time $\Theta(n^3)$. This naïve algorithm is fast enough if you implement it efficiently in Java or C++ but it is kind of borderline. My C++ implementation would take 4 seconds on the largest test and my Java implementation takes 5.5 seconds. I have *not* used any tricks in either program so it is possible to optimize them to get the runtime a bit down.¹ I want to stress that a good implementation of this basic $\Theta(n^3)$ solution without any tricks in C++ or Java is enough to get full marks but if you want to know how we can improve on this algorithm (either because your implementation was too slow or perhaps because you did it in Python and that was too slow) read on.

Looking at the dynamic programming solution it is clear that these nested loops contribute the most to the runtime of the algorithm:

```
for(int i=2;i<=n;i++){
    ...
    for(int j=2;j<=m;j++){
        //We want to compute D[i][j].
        //Either we came here from last row or column:
        D[i][j] = max(D[i-1][j], D[i][j-1]) + A[i][j];
        //Or from same row but a column number which divided j:
        for(int l=1;l<j;l++)
            if((j % l)==0) //If l | j
                D[i][j] = max(D[i][j], D[i][l] + A[i][j]);
    }
}
```

¹Here by trick I mean an optimization which is not algorithmic in nature but rather relies on changing a line or two that makes the program run faster. There is such a trick which makes the supplied C++ solution run 2 times faster; can you figure it out?

Here we are trying to compute $D[i][j]$ and we end up going over all l and checking if j is a multiple of l and if it is we will use $D[i][l]$ as a possible value to compute $D[i][j]$. This clearly takes $\Theta(n^3)$ but is there a better way to do it? In particular we could first go over all j and l once and find out for each j which are the l such that j is a multiple of l and store all of these in an array, lets call this $divisors[j]$. This way when computing $D[i][j]$ we only need to go over the l that are in $divisors[j]$. This is implemented in the `PA2sol-fast.cc` file on the webpage whose relevant parts are seen below:

```

...
for(int j=0;j<=m;j++) divisors[j].clear();
for(int l=1;l<=m;l++)
    for(int k=2;l*k<=m;k++)
        divisors[l*k].push_back(l);
...
for(int i=2;i<=n;i++){
    ...
    for(int j=2;j<=m;j++){
        //We want to compute D[i][j].
        //Either we came here from last row or column:
        D[i][j] = max(D[i-1][j], D[i][j-1]) + A[i][j];
        //Or from same row but a column number which divided j:
        for(size_t k=0;k<divisors[j].size();k++)
            D[i][j] = max(D[i][j], D[i][divisors[j][k]] + A[i][j]);
    }
}

```

The runtime of this algorithm (both asymptotic and in practice) of course depends on how big these $divisors[j]$ arrays end up being. If $divisors[j]$ ends up having around j numbers then this optimization is not worth it and might even make our algorithm much slower because of the extra cost of putting all these numbers in an array and the extra array lookup. On the other hand if $divisors[j]$ ends up having far fewer numbers then the optimization will make our program run much faster by computing these numbers once and using them later.

In particular the runtime is n times the total size of the divisors arrays, i.e. $n \sum_{j=1}^m |divisors[j]|$. In other words the question we have to be asking ourselves is how many divisors does a typical number between 1 and m have? This might sound like a hard question but the answer is not that hard, all one really needs to do is ask the question the other way around: How many multiples between 1 and m does a number l between 1 and m have? The answer to this second question is pretty simple: around $\lfloor n/l \rfloor$. So,

$$n \sum_{j=1}^m |divisors[j]| = n \sum_{l=1}^m \#\{j : l \in divisors[j]\} = n \sum_{l=1}^m \left\lfloor \frac{n}{l} \right\rfloor \leq nm \sum_{l=1}^m \frac{1}{l}.$$

So we have reduced our question (of analyzing this optimized solution) to the problem of computing (or approximating) the sum $\sum_{l=1}^m \frac{1}{l}$. This last sum might look scary and some of the usual tricks to compute a closed form for it will fail, however its value is a standard mathematical quantity called the m th *Harmonic number*. It is known that

$$\sum_{l=1}^m \frac{1}{l} = \ln m + \gamma + o(1),$$

where $\ln m$ is the natural logarithm of m (logarithm in base e) and γ is a constant around 0.58 called the *Euler-Mascheroni constant*.

To make the long story short we have,

$$n \sum_{j=1}^m |\text{divisors}[j]| \leq nm \sum_{l=1}^m \frac{1}{l} \leq nm \ln m.$$

So this implementation of the dynamic programming solution runs in time $O(n^2 \log n)$ which is much faster than $\Theta(n^3)$. In practice for the range of n in our problem this solution is about five times faster than the previous one.