COP 3502: Computer Science I
Spring 2004

– Day 8 –
Algorithm Analysis – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

Big-Oh Notation

- Used to represent the growth rate of a function.
- Allows algorithm designers to establish a relative order among functions by comparison of their dominant terms.
- Denoted as $O(N^2)$, read as "order N squared".

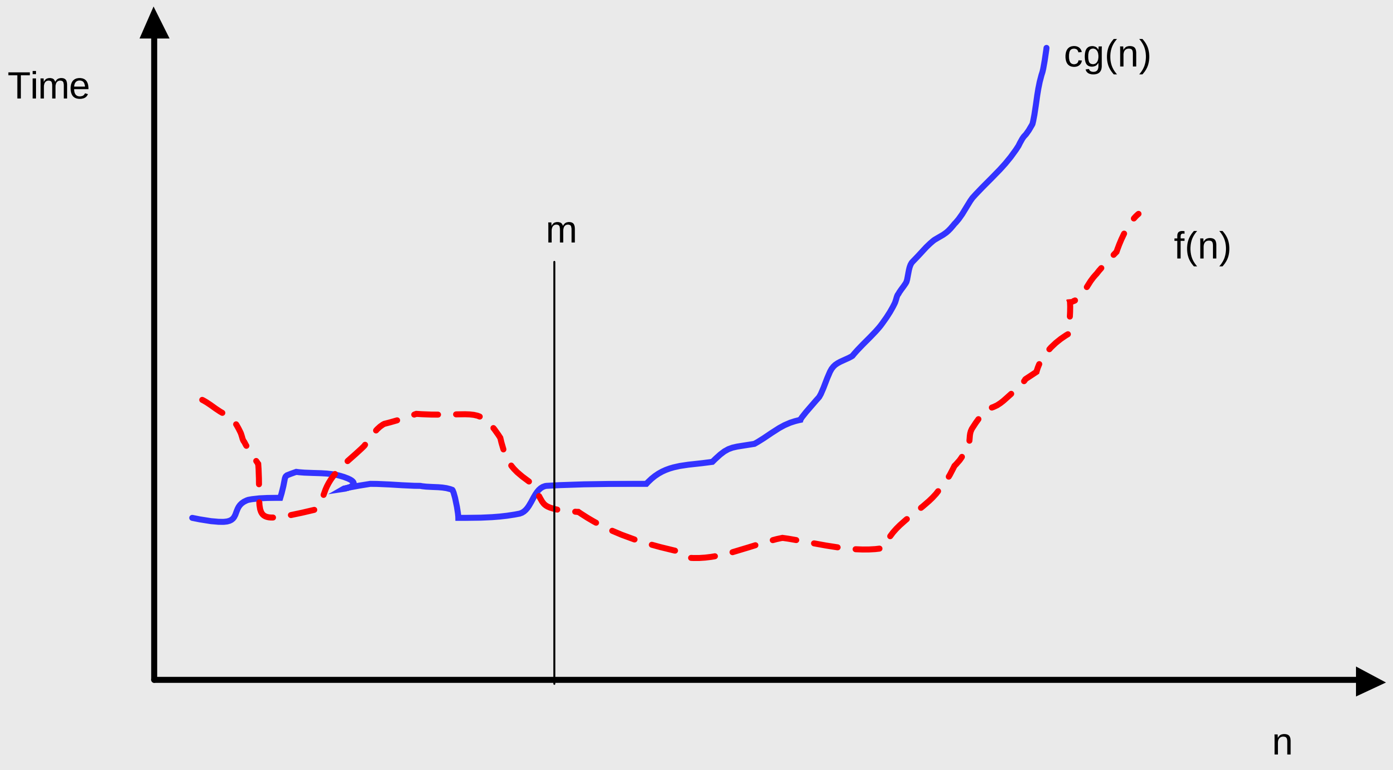
- constant function – $O(1)$
- logarithmic func. – $O(\log N)$
- log-squared func. – $O(\log^2 N)$
- linear func. – $O(N)$
- N log N func. – $O(N \log N)$
- quadratic func. – $O(N^2)$
- cubic func. – $O(N^3)$
- exponential func. – $O(2^N)$

Big-Oh Notation (cont.)

- Notation: f(n) = O(g(n)) [read as f(n) is big-oh of g(n)] means that f(n) is asymptotically smaller than or equal to g(n).

- Meaning: g(n) establishes an upper bound on f(n). The asymptotic growth rate of the function f(n) is bounded from above by g(n).

Big-Oh Notation (cont.)



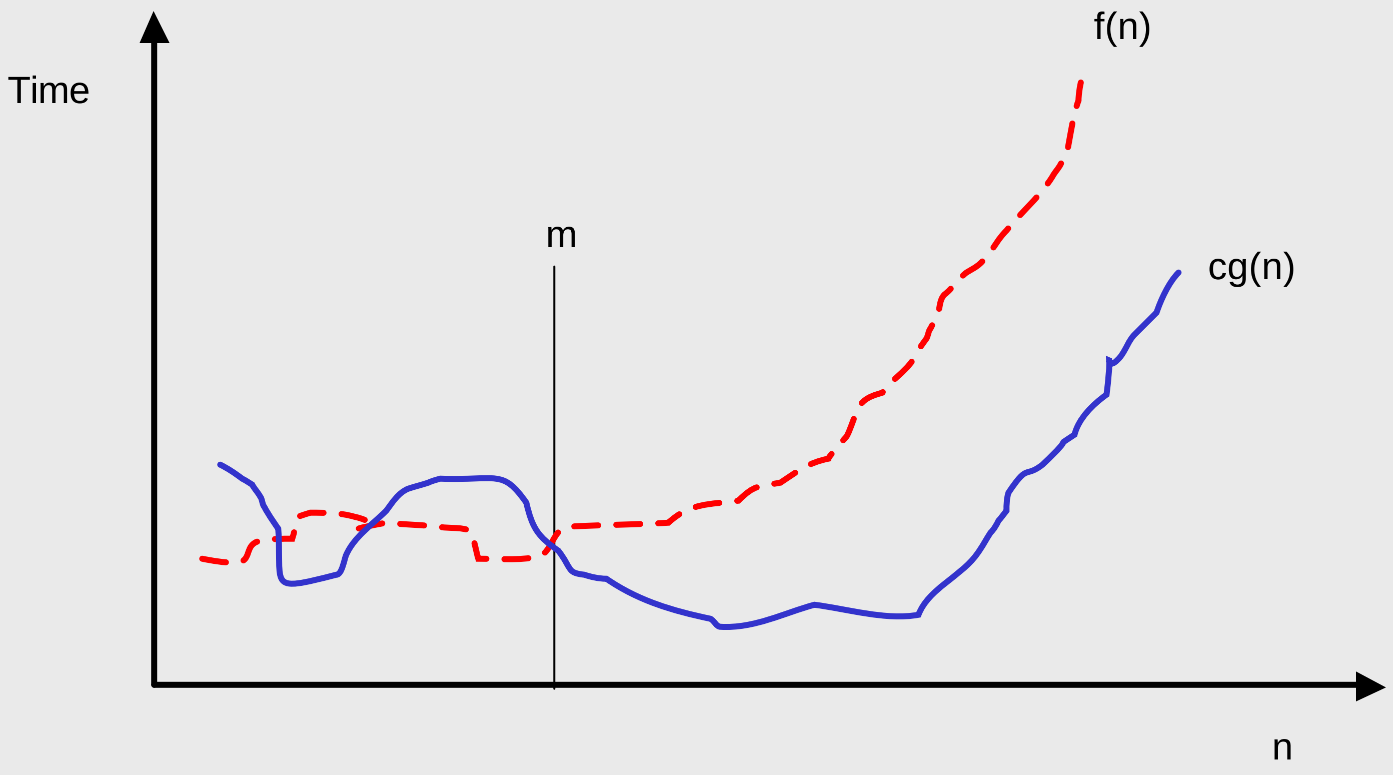
g(n) is an upper bound on f(n)

Omega Notation

Notation: f(n) = Ω(g(n)) [read as f(n) is omega of g(n)] means that f(n) is asymptotically bigger than or equal to g(n).

Meaning: g(n) establishes a lower bound on f(n). The asymptotic growth rate of the function f(n) is bounded from below by g(n).

Omega Notation (cont.)



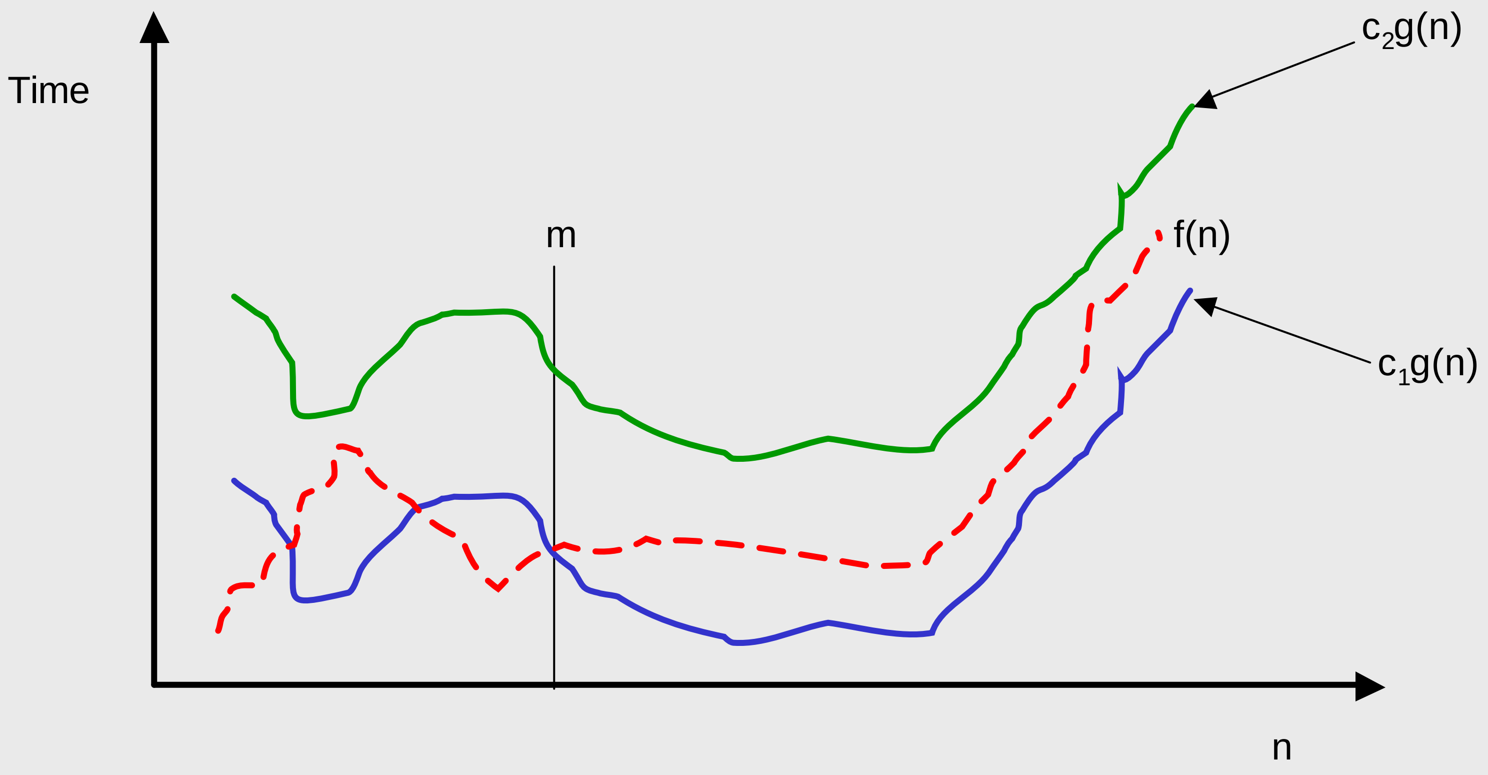
g(n) is an lower bound on f(n)

Theta Notation

Notation: $f(n) = \Theta(g(n))$ [read as f(n) is theta of g(n)] means that f(n) is asymptotically equal to g(n).

Meaning: g(n) and f(n) have the same characteristics.

Theta Notation (cont.)



g(n) is an upper and lower bound on f(n)

Little-Oh Notation

- Little Oh notation, o(f(n)), is commonly used in step count analysis and provides a strict upper bound on the asymptotic growth rate of the function.
- This means that f(n) is o(g(n)) iff f(n) is asymptotically smaller than g(n). [Note that the equal to case provided by the Big-Oh notation is not present in Little-Oh notation, thus the strict upper bound.]
- Definition:

$$f(n) = o(g(n)) \text{ iff } f(n) = O(g(n)) \text{ and } f(n) \neq \Omega(g(n)).$$

Growth Rates of Various Functions

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4.294×10^9
≈ 5.3	40	≈ 212	1600	64000	1.099×10^{12}
6	64	384	4096	262,144	1.844×10^{19}

Growth rate of various functions (in terms of the number of steps in the algorithm)

Example Growth Rates

- Let's assume that the functions shown in the table are to be executed on a machine which will execute a million instructions per second.
- A linear function which consists of one million instructions will require one second to execute.
- This same linear function will require only 4×10^{-5} seconds (40 microseconds) if the number of instructions (a function of input size) is 40.
- Next, let's consider an exponential function.

Example Growth Rates (cont.)

- When the input size is 32 approximately 4.3×10^9 steps will be required (since $2^{32} = 4.29\times10^9$).
  - Given our system performance this algorithm will require a running time of approximately 71.58 minutes.
- Now consider the effect of increasing the input size to 40, which will require approximately $1.1\text{x}10^{12}$ steps (since $2^{40} = 1.09\text{x}10^{12}$).
  - Given our conditions this function will require about 18325 minutes (12.7 days) to compute.
- If n is increased to 50 the time required will increase to about 35.7 years. If n increases to 60 the time increases to 36558 years and if n increases to 100 a total of $4\text{x}10^{16}$ years will be needed!

Algorithm Analysis

- Remember, that baring some type of looping statement, the code in a program (algorithm) is executed sequentially on von Neumann type architecture.
- This means that without loops, each statement is executed exactly one time and the running time of the algorithm is very easy to establish.
- Loops cause iteration and iteration increases the running time depending on how much iteration occurs.
- Therefore, we need to know, for the statements inside the loop, how many times they are executed.

Algorithm Analysis (cont.)

- Consider the two code segments shown below:

```
grandtotal = 0;
for (k = 0; k < n – 1; ++k) {
    rows[k] = 0;
    for (j = 0; j < n – 1; ++j) {
        rows[k] = rows[k] + matrix[k][j];
        grandtotal = grandtotal + matrix[k][j];
    }
}
```

$2n^2$

Code segment #1

```
grandtotal = 0;
for (k = 0; k < n – 1; ++k) {
    rows[k] = 0;
    for (j = 0; j < n – 1; ++j)
        rows[k] = rows[k] + matrix[k][j];
    grandtotal = grandtotal + rows]k];
}
```

$n^2 + n$

Code segment #2

- What is the total number of addition operations (in terms of *n*) performed by these two code segments?

Algorithm Analysis (cont.)

- Assume that we are working with a hypothetical computer that requires 1 microsecond (10^{-6}) seconds to perform an addition.
- If the value of $n = 1000$ then segment #1 would require 2 seconds to execute since:

$$[(2\times(1000)^2)\text{inst}] \times 10^{-6}\ \text{sec/inst} = 2\ \text{seconds}$$

- On the other hand, segment #2 would require just over 1 second since:

$$[(1000)^2 + 1000]\ \text{inst} \times 10^{-6}\ \text{sec/inst} = 1.001\ \text{seconds}$$

- If the value of n is increased to 100,000 then code segment #1 would require about 6 hours and code segment #2 would require about 3 hours.

Algorithm Analysis (cont.)

- Suppose that an algorithm takes T(N) time to run for a problem of size N – the question becomes – how long will it take to solve a larger problem? (Remember our earlier discussion of growth rates.)
- As an example, assume that the algorithm is an $O(N^3)$ algorithm. This implies: $T(N) = cN^3$.
- If we increase the size of the problem by a factor of 10 we have: $T(10N) = c(10N)^3$. This gives us:

$$T(10N) = 1000cN^3 = 1000T(N) \text{ (since } T(N) = cN^3\text{)}$$

  - Therefore, the running time of a cubic algorithm will increase by a factor of 1000 if the size of the problem is increased by a factor of 10. Similarly, increasing the problem size by another factor of 10 (increasing N to 100) will result in another 1000 fold increase in the running time of the algorithm (from 1000 to 1×10^6). $T(100N) = c(100N)^3 = 1\times10^6cN^3 = 1\times10^6T(N)$

Algorithm Analysis (cont.)

- A similar argument will hold for quadratic and linear algorithms, but a slightly different approach is required for logarithmic algorithms. These are shown below.
  - For a quadratic algorithm, we have $T(N) = cN^2$. This implies: $T(10N) = c(10N)^2$. Expanding produces the form: $T(10N) = 100cN^2 = 100T(N)$. Therefore, when the input size increases by a factor of 10 the running time of the quadratic algorithm will increase by a factor of 100.
  - For a linear algorithm, we have $T(N) = cN$. This implies: $T(10N) = c(10N)$. Expanding produces the form: $T(10N) = 10cN = 10T(N)$. Therefore, when the input size increases by a factor of 10 the running time of the linear algorithm will increase by the same factor of 10.
- In general, an *f*-fold increase in input size will yield an f^3-fold increase in the running time of a cubic algorithm, an f^2-fold increase in the running time of a quadratic algorithm, and an *f*-fold increase in the running time of a linear algorithm.

Algorithm Analysis (cont.)

- The analysis for the linear, quadratic, cubic (and in general polynomial) algorithms does not work in the presence of logarithmic terms.
- When an O(N logN) algorithm experiences a 10-fold increase in input size, the running time increases by a factor which is only slightly larger than 10.
  - For example, increasing the input by a factor of 10 for an O(N logN) algorithm produces: $T(10N) = c(10N) \log(10N)$.
  - Expanding this yields: $T(10N) = 10cN \log(10N) = 10cN \log N + 10cN \log N = 10T(N) + c'N$ (where $c' = 10c\log 10$).
  - As N gets very large, the ratio $T(10N)/T(N)$ gets closer to 10 (since $c'N/T(N) \approx (10 \log 10)/\log N$ gets smaller and smaller as N increases).
- The above analysis implies, for a logarithmic algorithm, if the algorithm is competitive with a linear algorithm for a sufficiently large value of N, it will remain so for slightly larger N.

How Much "Better" is O(log n) than O(n)

O(n)	O(log n)
16	4
64	6
256	8
1024 (1 kilo)	10
16,384	14
131,072	17
262,144	18
524,288	19
1.048,576 (1 Meg)	20
1,073,741 (1 Gig)	30

Comparison of O(n) to O(log n)

Comparison of O(log n) and O(n^2)

n	O(log n)	O(n^2)
16	4	256
64	6	4K
256	8	64K
1024	10	1M
16,384	14	256M
131,072	17	16 G
262,144	18	6.87×10^{10}
524,288	19	2.74×10^{11}
1.048,576	20	1.09×10^{12}
1,073,741	30	1.15×10^{18}

Comparison of O(log n) and O(n^2)

Estimating Run-time

- We know that we can't accurately compare run times measured on different machines, or with different operating systems or languages or compilers.
- What if we have measured the run time of a specific algorithm, on a specific combination of hardware, OS, language and compiler? How do we estimate the new run time just for a change in data size?
- If the algorithm is linear, i.e., O(n), it should be easy. If we know the ratio of old data size to new data size, we know the increase or decrease in time. For example: if it took 10 seconds for n = 50, then if we double the data size to n = 100, it should take twice as long.

Estimating Run-time (cont.)

- Since increases in data size are not always going to be by integer multipliers, we should generalize this calculation to a simple formula that can be used for any value of the data size.

$$\boxed{\frac{\text{old } n}{\text{old time}} = \frac{\text{new } n}{\text{new time}}}$$

- This formula is based on the fact that the ratio of data size (n) over time is the same for both the old and new data sizes, i.e., the time to perform each step is the same. The increase or decrease in time comes from the change in the size of the data, which means more or fewer steps are needed.

Estimating Run-time (cont.)

- Using this formula we can accurately predict the running time of our algorithms in terms of the input size.

Example:

If we know that a certain O(n) algorithm takes 30 seconds when n = 75, we can calculate the estimated time for n = 100 as follows:

$$\boxed{\frac{75}{30} = \frac{100}{t} \quad \Rightarrow \quad 2.5\,t = 100 \quad \Rightarrow \quad t = 40\text{s}}$$

Estimating Run-time (cont.)

- If the algorithm is $O(n^2)$, we'll need to modify our formula a little. Since the formula puts *n* over time, it is really calculating how much time each step takes, i.e., 75steps/30seconds = 2.5 steps/second or 2/5 seconds per step.

- So, what does that mean for our formula for $O(n^2)$?

- We know that an O(n) algorithm takes (by our definition of Order) roughly 1 step for each item in the data set, thus it takes n steps for n items. But, an $O(n^2)$ algorithm takes n^2 steps for n items. Thus, we should use that number of steps on top of the formula:

$$\boxed{\frac{(\text{old } n)^2}{\text{old time}} = \frac{(\text{new } n)^2}{\text{new time}}}$$

Estimating Run-time (cont.)

- Using this formula we can accurately predict the running time of our algorithms in terms of the input size.

Example:

If we know that a certain $O(n^2)$ algorithm takes 50 seconds when n = 10, we can calculate the estimated time for n = 20 as follows:

$$\frac{(10)^2}{50} = \frac{(20)^2}{t} \quad \Rightarrow \quad 2\,t = 400 \quad \Rightarrow \quad t = 200s$$

Estimating Run-time (cont.)

- Notice that our formula can also be used to answer a related question concerning the growth rate related to our algorithm by solving for the problem size rather than time.

Example:

If we know that a certain O(n^2) algorithm takes 50 seconds when n = 10, what is the largest problem instance that can be solved in two and a half minutes?

$$\boxed{\frac{(10)^2}{50} = \frac{(n)^2}{150} \quad \Rightarrow \quad n^2 = 300 \quad \Rightarrow \quad n \approx 17.3 \quad \Rightarrow \quad n = 17}$$

Estimating Run-time (cont.)

- What about exponential algorithms ($O(2^n)$)?
- Again, we must find the number of steps and divide by the time:

$$\boxed{\frac{2^{(\text{old } n)}}{\text{old time}} = \frac{2^{(\text{new } n)}}{\text{new time}}}$$

- Example: for a given O(2n) algorithm with n = 4 that takes 48 seconds, if we increase n to 6, how much longer will it take?

$$\boxed{\frac{2^{(4)}}{48} = \frac{2^{(6)}}{t} \quad \Rightarrow \quad 16\,t = 48 \times 64 \quad \Rightarrow \quad t = 192\,s}$$

Practice Problems Estimating Run-time

- Below are a few practice problems dealing with run-time or problem size estimations. Try these before you look at the answers on the next page.

1. An $O(n!)$ algorithm with a problem instance of size n= 4 requires 72 seconds to solve. How long will it take to solve a problem instance of size n = 5?
2. An $O(2^n)$ algorithm, a problem instance of size n = 7 required 96 seconds to solve. If you solved a different problem instance and the algorithm required 12 seconds to execute, how big was the problem instance?
3. An $O(n/\log_2 n)$ algorithm, a problem instance of size n = 32 runs in 96 msec. How long will the algorithm require if the problem instance is size n = 64?
4. An $O(n^2)$ algorithm executing on a problem instance of size n = 30 executes in 250 seconds. How large of a problem could be executed in 100 seconds?

Solutions to Practice Problems

- Below are the solutions to the practice problems on the previous page.

(1) $$\frac{4!}{72} = \frac{5!}{t} \Rightarrow t = \frac{(5!)\times 72}{4!} = 360\,\text{ms}$$

(2) $$\frac{2^7}{96} = \frac{2^n}{12} \Rightarrow 2^n = \frac{2^7 \times 12}{96} \Rightarrow 2^n = 16 \Rightarrow n = 4$$

(3) $$\frac{32/\log_2 32}{96} = \frac{64/\log_2 64}{t} \Rightarrow \frac{32/5}{96} = \frac{64/6}{t} \Rightarrow t = \frac{768}{64} = 12\,\text{s}$$

(4) $$\frac{30^2}{250} = \frac{n^2}{100} \Rightarrow n^2 = \frac{9000}{25} = 360 \Rightarrow n \approx 18.97 = 18$$