Big-O notations

How to you discuss run-time of an algorithm?

- Should we use actual time? But then 1) what machine do we use to measure the time, 2) what input size do we use? Note that different architectures, memory size, and many other computer characteristics effect the actual tun-time, so that the information will be useless on a different machine.
- Should we use actual assembly instruction count? Even then, due to pipelining the actual time will be different on another machine. Also different compiler may compile to a different assembly.

We need a more abstract solution that hides all of the above differences. Usually the approach is

1. Look at the algorithm – there are many different things happening: additions, multiplications, assignments, comparisons, etc. Choose operation that is most **representative** of the algorithm and just count that instruction.
2. Since most algorithms have `if`-statements, the count will be different depending on the input – even if 2 inputs have the same size, due to the actual values the counts may be different. To address this we will be providing 3 counts:
   - **best**-case scenario, the minimum count over all possible inputs of size n
   - **worst**-case scenario, the maximum count over all possible inputs of size n
   - **average**-case scenario, the average count over all possible inputs of size n
3. Another complication is that the count may be coming from 2 different parts (think `while`-loops) – which one to use? We will use the most "expensive" that accounts for the most of the count.
4. And the last – if our final count has a constant coefficient we will drop it. The reason is that we want a simplified count that is independent of hardware, compiler, and language. So our count is not the actual run-time, but rather an indicator of how much slower the algorithm will be if we increase input size. The latter number does not depend on the constant coefficient – it will be cancelled: n^2 will quadruple if n is doubled, same as $2n^2$, same as $1000 * n^2$.

Note: best-case is rarely useful.

Note: depending on the application, either worst or average may be more important. Example – real-time application (when it is crucial to have a guarantee that a process finishes in a particular amount of time), the worst-case should be used. For a web-server or any other application that executes something many times, and only the total amount of time is important, the average-case makes more sense.

Example:

```
void init( int * a, int n )
{
    // assume a is a valid pointer to memory of size of at least n integers
    for ( int i=0; i<n; ++i ) {
        a[i] = i+1;
    }
}
```

Naturally the size of the array is the input size. Operation involved are comparison, increment, addition, index, and assignment. We will choose assignment as the most representative of the purpose of the algorithm (initialization).

The `for`-loop obviously has n iterations, since the body of the loop has no `if`-statements, there will always be exactly n iterations, so best=worst=average=O(n).

Example:

```
void init( int ** a, int n )
    for (int i = 0; i < n; ++i) {       // n iterations
        for (int j = 0; j < n; ++j) {   // n iterations
            a[i][j] = i * j;            // n^2 multiplications (assignments)
        }
    }
}
```

Each of the n outer loop iterations has n inner loop for a total of n^2 assignments. Since the body of the loop has no `if`-statements, there will always be exactly n^2 iterations, so best=worst=average=O(n^2).

Example:

```
int** init( int ** a, int n ) // copy 2-dimensional array
    int ** p    = new int*[n];  // row pointers
    int *  data = new int[n*n];

    for ( int i=0; i<n; ++i ) {
        p[i] = data + i*n*sizeof( int ); // init row pointers
    }

    //actually copy data
    for (int i = 0; i < n; ++i) {       // n iterations
        for (int j = 0; j < n; ++j) {   // n iterations
            p[i][j] = a[i][j];          // n^2 multiplications (assignments)
        }
    }
    return p;
}
```

There are two loops, one has n iteration, the other is n^2 iterations, so Since the bodies of the loops has no `if`-statements, there will always be exactly $n+n^2$ iterations, so best=worst=average=O($n+n^2$)=O(n^2), since n^2 is growing faster.

Example:

```
void init( int ** a, int n ) // initializing ragged (triangular) array
    for (int i = 0; i < n; ++i) {   // n iterations
      for (int j = i; j < n; ++j) { // n, n-1 ,..., 1 iterations
          a[i][j] = (i == j) ? 1:0;
      }
    }
}
```

During the first iteration of the outer loop (i=0) there will be n iterations of the inner,
then n-1 iterations of the inner,
then n-2 iterations of the inner,
....
there will be 0 iterations of the inner loop when i=n-1 (which means we should have said `for (int i = 0; i < n-1; ++i)` for the outer loop.

So the total is

$$n + (n-1) + (n-2) + \ldots + 3 + 2 + 1$$

or

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) + n$$

this is an example of *arithmetic series*. Each next term of the sequence is the previous added a constant. The formula for the sum of the first n terms of arithmetic series is:

$$a_1 + \ldots + a_n = \frac{n}{2}(a_1 + a_n)$$

the idea behind the formula is very simple (supposedly 12-year old Carl Friedrich Gauss figured it out). Since each next term differs from the previous by the same amount, first + last is the same as second + one before the last and so on. That forms exactly $n/2$ pairs, each of which is $a_1 + a_n$.

Applying it to our arithmetic series

$$1 + 2 + \ldots (n-1) + n = \frac{n}{2}(n+1) = \frac{n^2}{2} + \frac{n}{2}$$

For the first time the run-time is represented by a sum of two functions, as stated before we will choose the most expensive one as a representative. Since n^2 grows much faster than n, we choose the $\frac{n^2}{2}$ term. Notice that $\frac{1}{2}$ is a constant coefficient, so our final result is $O(n^2)$. Since the code does not have conditionals, the number of iterations is the same for all runs, so best=worst=average=$O(n^2)$.

Example:

```
void find( int * a, int n, int val ) // linear search val in a
{
    for ( int i=0; i<n; ++i ) {
        if ( a[i] == val ) {
```

```
            return i; // found
        }
    }
    return n; // not found, return one past the last
}
```

Choose operation, since it is a find-style algorithm, the comparison is the most representative.

Notice that we have an `if`-statement, so the number of comparisons will be different depending on where *val* happens (or not) inside the array.

- if *val* is at the first position, it will be found after one comparison.
- if *val* is at the position i, it will be found after i comparisons.
- if *val* is at the last position, it will be found after n comparisons.
- if *val* is NOT in the array, algorithm will know that after n comparisons.

From above minimum number of comparisons is 1, giving us best case scenario $O(1)$ – constant time, does not depend on the size of the input. Really, if the element we are searching is at the first position, it will be found in the same time whether array is size 10 or 100000000.

Maximum (last position or not in the array) is n comparisons, so worst case scenario is $O(n)$.

To find average one has to list all possible scenarios with corresponding frequencies. Example calculating average: say you have a box with cards, each card has a number on it, find the average value of a card picked at random. If the values of the cards are 1, 2, and 3, then the average is

$$\frac{1+2+3}{3} = 2$$

that is assuming that there is exactly one card of each denomination.

If the frequencies of the cards are different, we have to use *weighted* average: say we have 3 cards with 1, 4 cards with 2, and 7 cards with 3

$$\frac{3*1+4*2+7*3}{3+4+7} = \frac{32}{14} = \frac{16}{7}$$

Back to the `find` algorithm:

	frequency	run-time
`val` is at position 0	1	1
`val` is at position 1	1	2
`val` is at position 2	1	3
.....		
`val` is at position n-2	1	n-1
`val` is at position n-1	1	n
`val` is NOT in the array	????	n

The question is what do we expect as the frequency of the last case.

Assume that it is 0, that is if we are seraching for an element, we have a guarantee that it is in the array, then

$$\frac{1*1+1*2+1*3+\ldots+1*(n-1)+1*n}{1+1+1+\ldots+1+1} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

If any integer number (assume 32-bit integer, so there are 2^{32} of them)

	frequency	run-time
`val` is at position 0	1	1
`val` is at position 1	1	2
`val` is at position 2	1	3
.....		
`val` is at position n-2	1	n-1
`val` is at position n-1	1	n
`val` is NOT in the array	$2^{32} - n$	n

Then the average is

$$\frac{1*1+1*2+1*3+\ldots+1*(n-1)+1*n+(2^{32}-n)*n}{1+1+1+\ldots+1+1+(2^{32}-n)} =$$

$$\frac{\frac{n(n+1)}{2}+(2^{32}-n)*n}{2^{32}} =$$

$$\frac{2^{32}*n-\frac{n^2}{2}+\frac{n}{2}}{2^{32}} =$$

$$n-\frac{n}{2}*\frac{n-1}{2^{32}}$$

Depending on n the above value will be almost $n/2$ if n is small (if $n = 2^{16}$, then average is $0.50000762939 * n$), if n is large then average will be almost n.

On the other hand using big-O both result in $O(n)$.

Example:

Binary search, assume array is sorted:

```
void find( int * a, int n, int val ) // binary search, assume a is sorted
{
    int end   = n;
    int begin = 0;

    while ( end-begin > 0 ) {
        if      ( a[ (begin + end) /2 ] > val ) { end   = (begin + end) /2; }
        else if ( a[ (begin + end) /2 ] < val ) { begin = (begin + end) /2; }
        else {
            return (begin + end) /2; // found
        }
    }
    return n; // not found, return one past the last
}
```

Note that during each iteration algorithm checked the middle value of the range and if it is the value we are looking for it returns, otherwise it decides which half (may be) contains the value – if middle element is greater then the value, then next range is left half, otherwise the right half. Note that after each iteration of the while-loop the size of the range that (possibly) contains the value is halfed.

Let's consider worst case scenario – value not found:

iteration	result	remaining size
1	not found	$n/2$
2	not found	$n/4$
3	not found	$n/8$
...	...	...
k	not found	$n/2^k$
...	...	...
last	not found	$n/2^{last} = 1$

The key equation is $n/2^{last} = 1$. The reason we know that is because we assumed this is the last iteration, after it `end-begin == 0` . The only positive number that produces 0 when devided by 2 is 1.

Solving $n/2^{last} = 1$ gives $last = \log_2 n$. So worst case scenario is $O(\log_2 n)$.

Similar to the previous example best case scenario is $O(1)$, and average is $O(\log_2 n)$.

example: find duplicates in an array

```
int a[] = { ... };  // unsorted array of integers
for (int i = 0; i < n; ++i) {   // n iterations
  for (int j = 0; j < n; ++j) { // n iterations
    if ( i != j && a[i] == a[j] ) return true;
  }
}
return false;
```

Best case $O(1)$ – when `a[0]==a[1]`.
Worst case $O(n^2)$ – when all values are different except for `a[n-2]==a[n-1]`.
Average – $O(n^2)$.
Same problem, but solution will presort the array:

```
int a[] = { ... };   // unsorted array of integers
sort( a );           // assume O(NlogN)
// after the sort equal elements will be next to each other
for (int i = 0; i < n-1; ++i) {    // n iterations
    if ( a[i] == a[i+1] ) return true;
}
return false;
```

Best case $N \log N + 1 = O(N \log N)$
Worst case $N \log N + N = O(N \log N)$
Average: $N \log N + N/2 = O(N \log N)$

example: algorithm with 2 parameters

Find values in an unsorted array M times, assume array size is n

Brute-force

```
int a[] = { ... };  // unsorted array of integers, size n
int v[] = { ... };  // array of values to find, size M
int count = 0;
for (int i = 0; i < M; ++i) {   // M iterations
  for (int j = 0; j < n; ++j) { // n iterations
    if ( a[j] == v[i] ) { ++count; goto br; } // do not care about duplicates
  }
br:
}
return count;
```

Run-time average=worst=$O(nM)$

Same problem, presort array:

```
int a[] = { ... };  // unsorted array of integers, size n
int v[] = { ... };  // array of values to find, size M
int count = 0;
sort(a); // assume n log n
for (int i = 0; i < M; ++i) {   // M iterations
    if ( binary_search( a, v[i] != n ) ++count; // log n
}
return count;
```

Run-time $= n \log n + M \log n = \log n(n + M)$.

Notice that comparison of $M * n$ and $(M + n) * \log n$ depends on values of n and M

- M is much smaller than n:

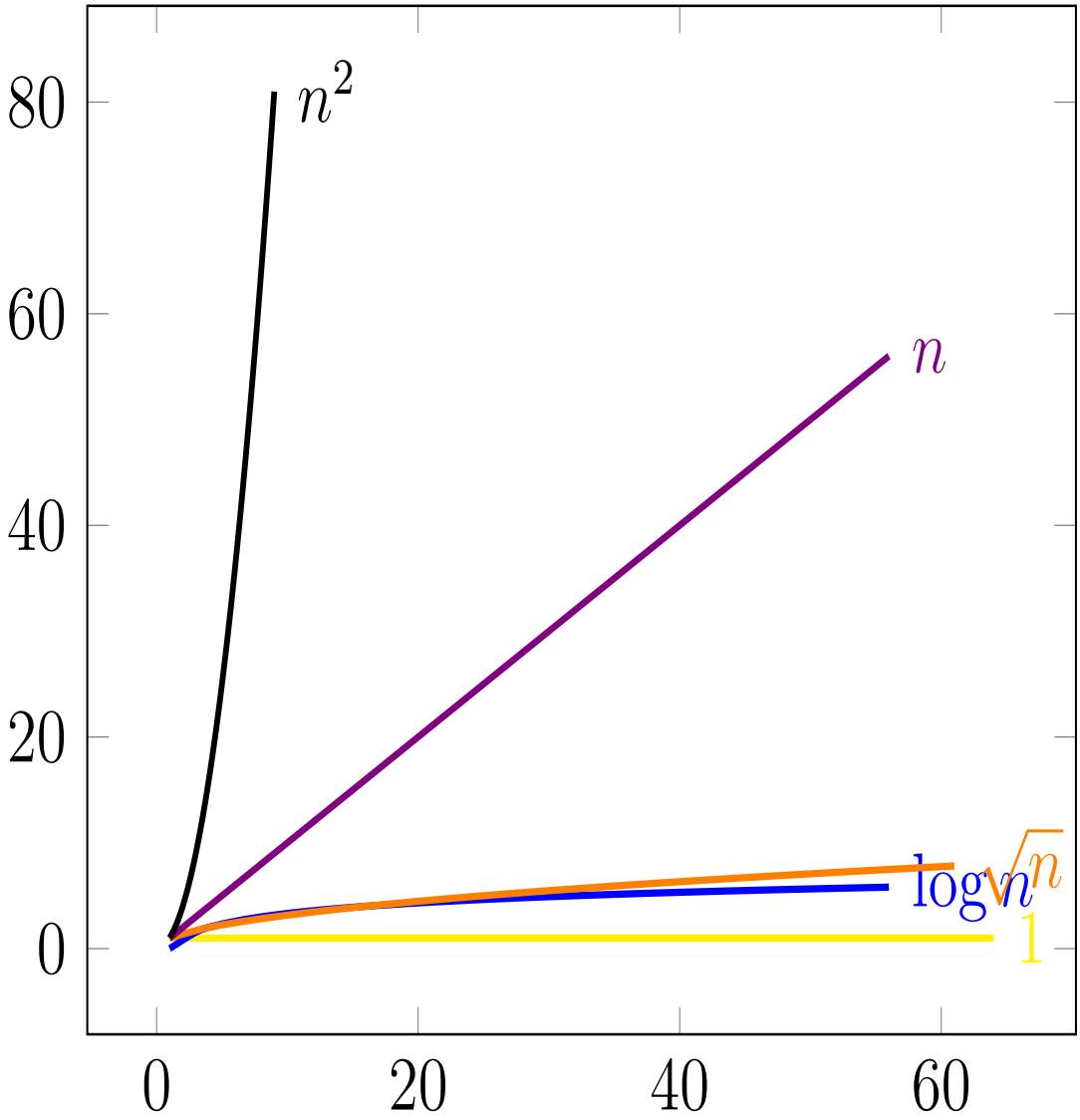
$$M * n = O(n) \text{ better}$$
$$(M + n) \log n = O(n \log n)$$

- M is comparable to n:

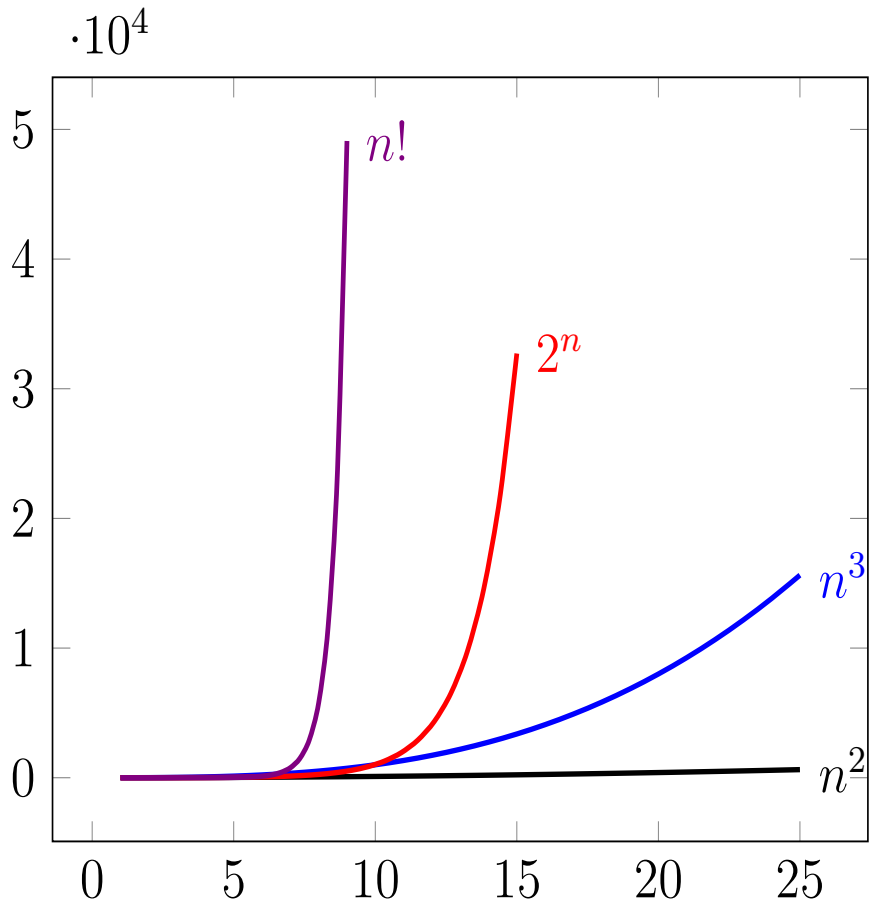
$$M * n = O(n^2)$$
$$(M + n) \log n = O(n \log n) \text{ better}$$

Most useful functions for big-O in order:

- Class $O(K)$ or $O(1)$. Constant $f(n) = K$, run-time does not depend on the size of the input. Examples: index operator in C++ array.
- Class $O(\log n)$. Logarithmic $f(n) = \log n$. Examples: binary search.
- Class $O(\sqrt{n})$. No special name. Examples: brute-force test for a number to be prime.
- Class $O(n)$. Linear $f(n) = n$. Examples: linear search.
- Class $O(n \log n)$. No special name. Examples: merge sort, heap sort, average case of quicksort.
- Class $O(n^2)$. Quadratic $f(n) = n^2$. Examples: bubble sort, selection sort, insertion sort.
- Class $O(n^3)$. Cubic $f(n) = n^3$. Examples: Floyd-Warshall algorithm.
- Class $O(K^n)$. Exponential $f(n) = K^n$, where K is a constant. Examples: almost everything! – depth-first search, breadth-first search, A*, satisfiability problem.
- Class $O(n!)$. Factorial $f(n) = n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$. Usually combined with exponential class.



Faster growing functions will look like a vertical line if I add them to the above plot, so drawing them separately. Note that n^2 (black) is in both graphs – biggest above and smallest below. The next graph uses different scale for horisontal (1..30) and vertical axes (0..55,000). Due to that n^2 looks almost flat.


$\cdot 10^4$
5
4
3
2
1
0
$n!$
2^n
n^3
n^2
0
5
10
15
20
25

Uses of big-O.

Compare algorithms: if some says that algorithm A is $O(n \log n)$ and algorithm B is $O(n^2)$, we should prefer algorithm A. But one has to remember that $O(n \log n)$ is faster than $O(n^2)$ for **big** values of n. How big – we do not know. What can happen is that you are interested in small values of n, you may still prefer algorithm B. Famous example is algorithm A = quicksort average runt-time $O(n \log n)$, algorithm B = insertion sort average runt-time $O(n^2)$. Quicksort will be faster for big values 0f $n > 20$, but insertion sort will be faster if $n \leq 8$. Obviously the numbers depend on the particular implementations.

Another way of using big-O is to **estimate run-time**. Say you have a $O(n^2)$ algorithm and you want to know how long will it take to run it on on input of size 1000. It may be something like this – you have to perform a calculation and you have 1 week to do so. You already have $O(n^2)$ implementation, question – will it be on-time. You do not want to start it and by the end of the week, when you do not have any time left, realize that it will not finish. Your plan is to estimate the run-time, if it is less thean a week, then start the computation, otherwise you have to implement a different algorithm.

So you may proceed like this: run on an input of size 100, say it takes 1 hour. Since runtime is $O(n^2)$, the actual function for the run-time is $k \times n^2$ where k is some unknown constant. The previous measurment gives us

$$k \times 100^2 = 1$$

To estimate runtime for $n = 1000$ we need to calculate

$$k \times 1000^2 = k \times (100 \times 10)^2 = k \times 100^2 \times 10^2 = \text{ [using measurment from above]} = 1 \times 100 = 100 \text{ hours}$$

which is about 4 days and 4 hours. Should be OK then.

Same problem but runtime was $O(n^3)$. To estimate runtime for $n = 1000$ we need to calculate

$$k \times 1000^2 = k \times (100 \times 10)^3 = k \times 100^3 \times 10^3 = \text{ [using measurment from above]} = 1 \times 1000 = 1000 \text{ hours}$$

which is more than a month. In this case we should start thinking about a new algorithm.

Polynomial run-times are easy to work with: given a $O(n^k)$ algorithm increasing input size by a factor of s increases run-time by a factor of s^k (the initial input size does not matter). Easy to prove – left as an exercise.