Today's class

- Analysis of algorithms
- Asymptotic notation

Analysis of algorithms

References

- Read Chapter-2 from Mark Allen Weiss
- Read Chapter-2,3 from CLRS
  - CLRS => CORMEN, LEISERSON, RIVEST, STEIN
- Read Chapter-1 from Narasimha Karumanchi

Analysis of algorithms

- How do we say that one algorithm is better than the other?

- Possible strategy
  - Implement both algorithms
  - Run both algorithms for various inputs
  - Analyse the runtime and memory usage of both implementations
  - What is the drawback of this approach?

Drawbacks

- Experiments are performed on a set of test inputs that may not include all possible behaviors
  - It's very hard to test for all possible kinds of inputs
    - Think about all possible inputs for a sorting algorithm
- The algorithm may perform differently on different platforms
  - Some CPUs are faster than others, e.g., Intel i7 vs. Intel i3
- The algorithm needs to be fully implemented before the runtime analysis

Analysis of algorithms

- Another strategy is to count the total number of CPU instructions that may execute for the worst input (i.e., the input for which the algorithm is expected to behave very badly)
  - What is a CPU instruction?

CPU instructions

```
d = a + b + c;
```

```
movl -4(%rbp), %eax
movl -8(%rbp), %ebx
movl -12(%rbp), %ecx
mov %eax, %edx
add %ebx, %edx
add %ecx, %edx
mov %edx, -16(%rbp)
```

A program statement is eventually converted to a sequence of CPU instructions that actually executes when you run an application.

CPU instructions

```
d = a + b + c;
```

```
movl -4(%rbp), %eax
movl -8(%rbp), %ebx
movl -12(%rbp), %ecx
mov %eax, %edx
add %ebx, %edx
add %ecx, %edx
mov %edx, -16(%rbp)
```

Analyzing the assembly is hard because high-level operations like loops that take most of the time are not intuitive in assembly.

Assembly code is the actual code that is going to execute at runtime. However, it is hard to analyze the assembly code than a C program. Therefore, instead of counting the number of CPU instructions, we focus on counting the high-level operations.

Analysis of algorithms

- Time taken by an algorithm is directly proportional to the sum of the costs of CPU instructions executed at runtime
  - Counting the number of CPU instructions may be a tedious job
- Alternatively, counting the number of high-level (primitive) operations is a good approximation

Primitive operations

- Assigning a value to a variable: a = b
- Calling a method: foo(a, b)
- Performing an arithmetic operation: a + b
- Comparing two numbers: a <= b
- Indexing into an array: a[b]
- Returning from a method: return a

Fibonacci numbers (algorithm-1)

- Recursive solution

n	Num operations
<=1	2
>1	7

```
1. int fib(int n) {
2.     if (n <= 1)
3.         return n;
4.     return fib(n-1) + fib(n-2);
5. }
```

If n <= 1, the comparison operator at line-2 and the return statement at line-3 take place, so the total number of operations is two. When n > 1, then the compare at line-2 is counted as one operation, computing n-1 and n-2 at line-4 are two operations, calling fib(n-1) and fib(n-2) are two operations, adding fib(n-1) and fib(n-2) is one operation and finally returning the sum at line-4 is one operation. Therefore the total number of operations, in this case, is 7.

Fibonacci numbers (algorithm-2)

- Iterative solution

n	Num operations
<=1	2
>1	4 + (n-1)*6 + 1

```
1. int fib(int n) {
2.   if (n <= 1) {
3.     return n;
4.   }

5.   int prev = 1;
6.   int pprev = 0;
7.   int res, i;

8.   for (i = 2; i <= n; i++) {
9.     res = prev + pprev;
10.    pprev = prev;
11.    prev = res;
12.  }
13.  return res;
14.}
```

In this algorithm, if n <= 1, the comparison and return operations at lines-2 and 3 execute. Otherwise, the total number of operations includes comparison at line-2; initializations at lines-5 and 6; one-time initialization of i in the for-loop at lines-8; "n-1" times execution of, 1> the compare and increment operations at line-8, 2> addition at line-9, and 3> assignments at lines-9, 10, 11; and finally the return operation at line-13.

Fibonacci numbers (algorithm-3)

```
// mul takes a matrix A and R as input
// returns the result of A^n in R
1. void mul(int A[2][2], int R[2][2], int n) {
2.   if (n == 1) {
3.     R[0][0] = A[0][0];  R[0][1] = A[0][1];
4.     R[1][0] = A[1][0];  R[1][1] = R[1][1];
5.     return;
6.   }
7.   if (n % 2 == 0) {
8.     mul(A, R, n/2);
9.     // mul2 takes two 2x2 matrices as input and
10.    // returns the multiplication in the first
11.    // matrix
12.    mul2(R, R);  // R <- R * R
13.  }
14.  else {
15.    mul(A, R, (n-1)/2);
16.    mul2(R, R);  // R <- R * R
17.    mul2(R, A);  // R <- R * A
18.  }
19.}
```

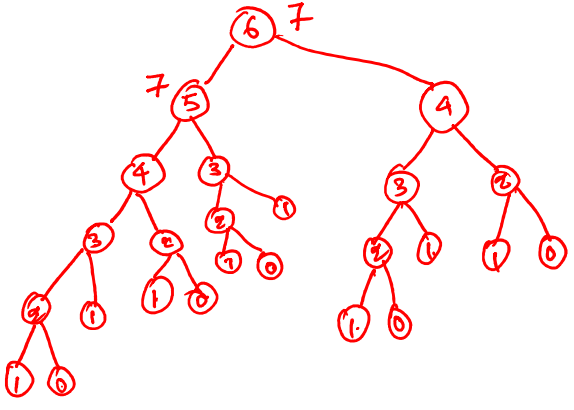
```
int fib(int n) {
  if (n <= 1) {
    return n;
  }
  int A[2][2];
  int R[2][2];
  A[0][0] = 1; A[0][1] = 1;
  A[1][0] = 1; A[1][1] = 0;
  mul(A, R, n-1);
  return R[0][0];
}
```

n	Num operations
=1	14
mul2	41
n%2 = 0	46
n%2 = 1	88
fib	13

In mul, if n=1, the total number of operations include the comparison at line-2, computation of the address of the array elements R[0][0], A[0][0], etc. at lines-3, 4, four assignments at lines-3,4, and finally the return operation at line-5. Otherwise, if n is even, the comparison at line-2 executes, followed by the computation of n%2 and the comparison operation at line-7, computation of n/2 and calling mul at line-8, and 41 operations corresponding to mul2. Else if n is odd, then in addition to the operations at lines-2,7, subtraction, divide, and call operations are performed at line-15, and calls to mul2 at lines-16,17 take 41 operations each.

Fibonacci numbers

- Algorithm-1 : n = 6



n	Num operations
<=1	2
>1	7

$13 \times 2 + 12 \times 7$

$26 + 84$

$= 110$

For the first algorithm, many recursive calls will take place. Since we have already computed the number of operations taken by the fib routine, it is easy to compute the total number of operations that will take place during the computation of fib(6), which is 110.

Fibonacci numbers

- Algorithm-2: n = 128

n	Num operations
<=1	2
>1	4 + (n-1)*6 + 1

4 + 127*6 + 1

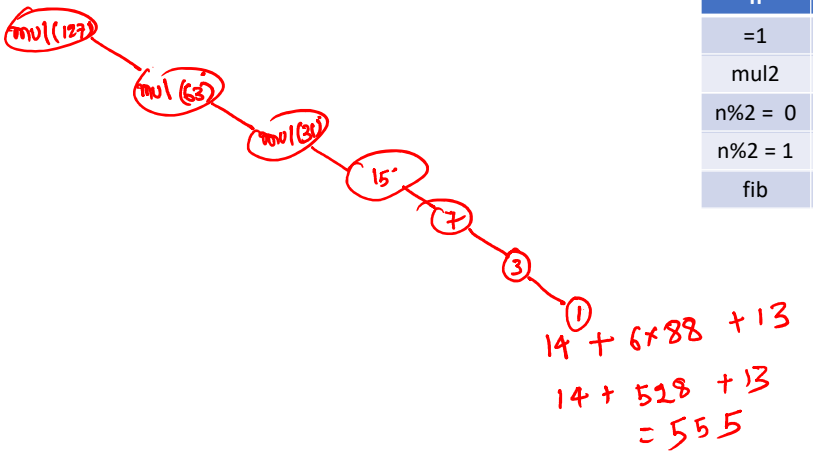
= 4 + 762 + 1

= 767

For the second algorithm, computing the number of operations is very easy because everything is done within the function. For n = 6, it does around 35 operations which is better than the previous one. For n=128, this algorithm performs 767 operations.

Fibonacci numbers

- Algorithm-3 : n = 128



n	Num operations
=1	14
mul2	41
n%2 = 0	46
n%2 = 1	88
fib	13

Let's look at the third algorithm. As we already know that for n = 128, mul(127) is invoked. mul(127) does seven recursive calls until mul(1) is reached. Since we have already computed the number of operations required for the mul routine, we can compute the total number of operations performed by this algorithm to compute fib(128), which is 555. The total number of operations in this algorithm is less than the iterative algorithm, as computed on the previous slide. This algorithm is better than the iterative one because, at every recursive call, it reduces the problem to half of the size. For example, if the value is 100000, the first recursive call brings it down to 50000, the second makes it 25000, and so on. Therefore, we can reach the base case (which is one) in very few recursive calls. Whereas for 100000, the iterative algorithm iterates from 2 to 100000, computing every intermediate Fibonacci number; thus, it does more operations.

Number of operations

- Algorithm-1 ($f(n) = f(n-1) + f(n-2)$)
  - $\left(7 * 2^{\frac{n-1}{2}}\right) - 5 \leq num\ operations \leq (7 * 2^{n-1}) - 5$
- Algorithm-2 (iterative)
  - $num\ operations = 6\text{n} - 1$
- Algorithm-3 (matrix multiplication)
  - $c_1 log_2 n \leq \text{num operations} \leq c_2 \log_2 \text{n}$
    - for some constants c_1 and c_2

We can represent the total number of operations as a function of input size n. We will discuss later in this course how to compute these bounds.

Time complexity

- The number of primitive operations in the worst case as a function of input size
  - e.g., the time complexity of the iterative algorithm for Fibonacci is 6n - 2

Time complexity

- Does counting the number of primitive operations give us the actual time complexity?

Time complexity

- Does counting the number of primitive operations give us the actual time complexity?
  - Does calling a function is same as a variable assignment?

- Can we simplify the analysis further?

Time complexity

- Does counting the number of primitive operations give us the actual time complexity?
  - Does calling a function is same as a variable assignment?

- Can we simplify the analysis further?
  - Instead of finding the real complexity, we try to find the lower and upper bounds on the total number of operations
  - We try to make lower and upper bounds close to the actual complexity

Time complexity

- We will discuss a general method to compute time complexity, which
  - Considers all possible inputs
  - Can be used to compare two algorithms in a way that is independent of underline hardware and software platform
  - Can be performed on the pseudo-code of an algorithm

Which algorithm is better?

The polynomial expressions correspond to the total number of operations needed for a given algorithm for input size n

$A = 3n^2 + 2$
$B = 8n^2 + 3n + 10$

Algorithm A is better because it performs fewer operations than algorithm B for all n >= 1.

Which algorithm is better?

$A = n^2$
$B = 10n + 1$

Which algorithm is better?

$A = n^2$
$B = 10n + 1$

- Time complexity matters for large input size
  - Even though for a smaller input (i.e., n <= 10), algorithm A is better, for all “n > 10”, algorithm B will perform the desired task in lesser numbers of operations

Which algorithm is better?

$A = n^2 + 1$
$B = n^2 + 1000n + 10000$

The first algorithm is faster than the second.

Which algorithm is better?

$A = n^2 + 1$
$B = n^2 + 1000n + 10000$

n	A = n^2 + 1	B = n^2 + 1000n + 10000	A/B
1000	1000001	2010000	0.497513
10000	100000001	1.1E+08	0.909008
100000	1E+10	1.01E+10	0.990098
1000000	1E+12	1E+12	0.999001
10000000	1E+14	1E+14	0.9999

For large n, only the highest order term in the time complexity matters.

Even though the first algorithm is faster than the second, the ratio of the number of operations performed by the first and second algorithms remains close to 1 for large values on n. From this observation, we can conclude that only the highest-order term in the time complexity matters.

Which algorithm is better?

$A = 3n^2 + 1$
$B = 7n^2 + 1000n + 10000$
$C = 100n^{1.5} + 2000n + 20000$

n	A = $3n^2 + 1$	B = $7n^2 + 1000n + 10000$	C = $100n^{1.5} + 2000n + 20000$	A/B	C/A
1000	3000001	8010000	5182277.7	0.374532	1.727425
10000	3E+08	7.1E+08	120020000	0.422529	0.400067
100000	3E+10	7.01E+10	3.362E+09	0.42796	0.112077
1000000	3E+12	7E+12	1.02E+11	0.42851	0.034
10000000	3E+14	7E+14	3.182E+12	0.428565	0.010608
100000000	3E+16	7E+16	1.002E+14	0.428571	0.00334

Let's compare these algorithms. The degree of polynomials A and B is 2, which is larger than C (1.5). Therefore, if we take the ratio of the total number of operations for algorithms C and A, it goes close to zero as the n increases. In this case, we can say that algorithm C is a huge improvement over algorithm A. However, if we compare the number of operations for A and B, the ratio remains close to 0.42, even for large values of n. In this case, even though algorithm A is a good improvement over B, it's not huge. In our analysis, we will focus on the huge improvement aspects and disregard the constant factors in the time complexity.

Which algorithm is better?

$A = 3n^2 + 1$
$B = 7n^2 + 1000n + 10000$
$C = 100n^{1.5} + 2000n + 20000$

n	A = $3n^2 + 1$	B = $7n^2 + 1000n + 10000$	C = $100n^{1.5} + 2000n + 20000$	A/B	C/A
1000	3000001	8010000	5182277.7	0.374532	1.727425
10000	3E+08	7.1E+08	120020000	0.422529	0.400067
100000	3E+10	7.01E+10	3.362E+09	0.42796	0.112077
1000000	3E+12	7E+12	1.02E+11	0.42851	0.034
10000000	3E+14	7E+14	3.182E+12	0.428565	0.010608
100000000	3E+16	7E+16	1.002E+14	0.428571	0.00334

Algorithm A is better than B. However, the improvement factor remains constant for large input sizes. In contrast, algorithm C is a huge improvement over algorithm A.

Efficiency of algorithms

- Observations so far
  - We analyze the algorithms for **large input size**
  - Only the **highest degree term** in the time complexity matters
  - We disregard the constant factors in the formula and focus on the **huge-improvement** aspects of the algorithm

What is huge improvement?

- An algorithm A is a huge improvement over algorithm B, if

$$\lim_{n \to \infty} \frac{time\ complexity\ of\ A\ (n)}{time\ complexity\ of\ B\ (n)} = 0$$

What is huge improvement?

- An algorithm A is a huge improvement over algorithm B, if

$$\lim_{n\to\infty} \frac{time\ complexity\ of\ A\ (n)}{time\ complexity\ of\ B\ (n)} = 0$$

$\frac{10000n}{n^2}$ $\frac{10000n}{n}$

$A = 10000n + 2000000$
$B = n^2$
Is the algorithm A is huge improvement over B?

Yes, because for large n, the ratio of A and B is close to zero.

What is huge improvement?

- An algorithm A is a huge improvement over algorithm B, if

$$\lim_{n\to\infty} \frac{time\ complexity\ of\ A\ (n)}{time\ complexity\ of\ B\ (n)} = 0$$

$A = n^2$
$B = 10000n^2 + 2000000$
Is the algorithm A is huge improvement over B?

No, because the ratio of A and B remains almost the same even for large values of n.

Asymptotic notation

(Captures the observations we have made so far)

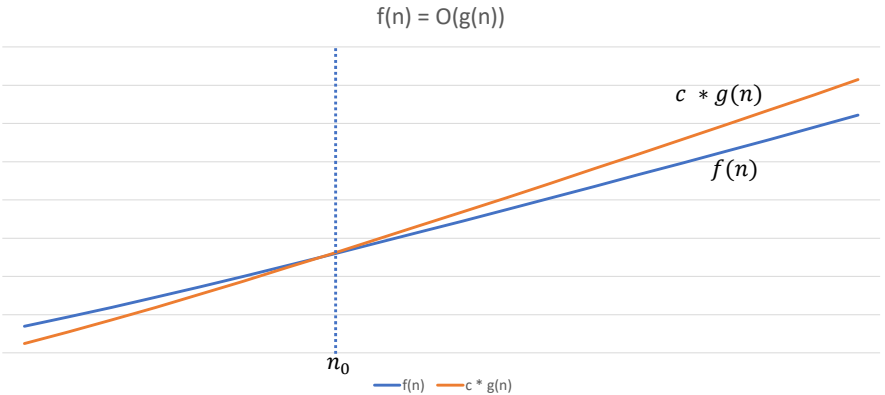
Big-Oh notation

Let $f(n)$ and $g(n)$ are increasing functions of n

$f(n)\ is\ O\big(g(n)\big)$ (pronounced as order of)
if for some constants $c > 0$ and $n_0 \geq 1$

$$f(n) \leq cg(n), \qquad for\ all\ n > n_0$$

Big-Oh notation



c * g(n) is an asymptotic upper bound for f(n)

Example

$8n^4 + 2n^3 + n^2 + 3 \;\; is \; O(n^4)$

$8n^4 + 2n^4 + n^4 + 3n^4$

$\leq 14\, n^4$

Because the value of "$8n^4 + 2n^3 + n^2 + 3$" is less than "$8n^4 + 2n^4 + n^4 + 3n^4$", i.e., $14n^4$, therefore the order is n^4.

Example

$a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_k n^k$ is $O(n^k)$ if $a_k > 0$

Example

$a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_k n^k$ is $O(n^k)$ if $a_k > 0$

The polynomial is less than: $(abs(a_0) + abs(a_1) + \cdots + abs(a_k)) * n^k$ for all $n \geq 0$, therefore, this is $O(n^k)$

Here, “abs” denotes the absolute value of the coefficients.

Example

$5n^2 + 8nlog_2n + 3$ $= O(n^2)$

$3nlog_2n + n$ $= O(nlogn)$

$10n + 3\log_2 n$ $= O(n)$

100 $= O(1)$

$100 * 2^{n+12}$ $= \leq 100 \times 2^{12} \times 2^n = O(2^n)$

2^{1000} $= O(1)$

Example

$5n^2 + 8nlog_2n + 3 \quad = O(n^2)$

$3nlog_2n + n \quad = O(n \log\ n)$

$10n + 3\log_2 n \quad = O(n)$

$100 \quad = O(1)$

$100 * 2^{n+12} \quad = O(2^n)$

$2^{1000} \quad = O(1)$

Big-Oh notation

- Big-Oh is used to describe asymptotic upper bound
- Even though there are many possibilities for the upper bound, we try to stay close to the real complexity
  - e.g., $3n^2 + 8\ is\ O(n^4)$ but doesn't make much sense
- We remove the constant factors and lower order terms in the big-Oh notation

Space complexity

- Similar to the time complexity, we can also measure space complexity
- The space complexity tells us about the memory utilization of the program in the worst case as a function of input size
- For most algorithms, space complexity is not a big concern

Analysis of algorithm

- Time complexity
  - Predict how much time an algorithm will take for a given input size
- Space complexity
  - Predict how much space an algorithm will take for a given input size

Linear search

```
int lsearch(int arr[], int val, int n) {
   int i;
   for (i = n-1; i >= 0; i--) {
      if (arr[i] == val)
         return i;
   }
   return -1;
}
```

Time complexity:

$C_1 * n + C_2$

$O(n)$

Linear search

```
int lsearch(int arr[], int val, int n) {
   int i;
   for (i = n-1; i >= 0; i--) {
      if (arr[i] == val)
         return i;
   }
   return -1;
}
```

Time complexity:

$$n * c_1 + c_2 = O(n)$$

This slide computes the number of operations performed by the lsearch routine. As discussed before, we are going to ignore the constant factors and compute the number of operations in the worst case. Let's assume that the body of the for loop does c_1 number of operations and the code outside the loop does c_2 number of operations. In the worst case, this loop may execute n number of times (e.g., when val is not present in the array). Therefore, the total number of operations in the worst case would be $n * c_1 + c_2$, which is O(n). So, in the worst case, this program may execute O(n) operations.

Selection sort

```
void selection_sort(int arr[], int n)
{
  int i, idx;

  for (i = 0; i < n - 1; i++) {
    idx = find_min(arr, i, n);
    if (idx != i) {
      swap(arr, i, idx);
    }
  }
}
```

$c_1 \times (end - start - 1) + c_2$

Time complexity:

$(c_1 \times (n-1) + c_2 + c_3) + (c_1 \times (n-2) + c_2 + c_3) + (c_1 \times (n-3) + c_2 + c_3) + \cdots + (c_1 + c_2 + c_3) + c_4$

```
int find_min(int arr[], int start, int end)
{
  int i, min, idx;

  min = arr[start];
  idx = start;
  for (i = start + 1; i < end; i++) {
    if (arr[i] < min) {
      min = arr[i];
      idx = i;
    }
  }
  return idx;
}

void swap(int arr[], int i, int j)
{
  int tmp = arr[i];
  arr[i] = arr[j];
  arr[j] = tmp;
}
```

Let's look at the find_min procedure. The for loop executes "end – start – 1" times. If the number of operations inside this loop is c_1, the for loop in the find_min will perform (start - end -1) * c_1 number of operations. Because, outside the for loop, find_min perform only a constant number of operations, say c_2. So the total number of operations performed by find_min is "c_1 * (start - end - 1) + c_2". The first time we call find_min from the selection sort, the value of (start, end) is (0, n). So, the first invocation of find_min performs "c_1 * (n - 1) + c_2" operations. The second time find_min is called with (start, end) as (1, n), so the number of operations in this step is "c_1 * (n - 2) + c_2". find_min is called until the value of (start, end) reaches (n-2, n), in which case, it performs "c_1 + c_2" operations. The rest of the for loop body in the selection_sort algorithm does a constant number of operations, say c_3. Because the for loop in selection sort executes n-1 number of times; therefore, in addition to the number of operations performed in find_min, it additionally performs "(n - 1) * c_3" operations. Outside the for loop, seclection_sort performs a constant number of operations, say c_4. Considering all of this, the number of operations performed by selection_sort is listed on this slide.

Time complexity

$$c_2(n-1) + c_3(n-1) + c_4 + c_1(1+2+\ldots+n-1)$$

$$= (n-1)(c_2+c_3) + c_4 + c_1 \times \frac{(n-1)\times n}{2}$$

$$= O(n^2)$$

..

Time complexity

$$
\begin{aligned}
T(n) = {} & (c_1(n-1)+c_2) + \\
& (c_1(n-2)+c_2) + (c_1(n-3)+c_2) + \ \ldots + (c_1 * 1 + c_2) + \\
& c_3(n-1) + c_4 \\
= {} & c_1(1+2+3+\ \ldots + (n-1)) + c_2(n-1) + c_3(n-1) + c_4 \\
= {} & c_1\left(\frac{(n-1)n}{2}\right) + c_2(n-1) + c_3(n-1) + c_4 \\
= {} & O(n^2)
\end{aligned}
$$

Fibonacci numbers

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }

  int prev = 1;
  int pprev = 0;
  int res, i;

  for (i = 2; i <= n; i++) {
    res = prev + pprev;
    pprev = prev;
    prev = res;
  }
  return res;
}
```

Time complexity:

$(n-1) \times c_1 + c_2$

$O(n)$

Fibonacci numbers

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }

  int prev = 1;
  int pprev = 0;
  int res, i;

  for (i = 2; i <= n; i++) {
    res = prev + pprev;
    pprev = prev;
    prev = res;
  }
  return res;
}
```

Time complexity:

$$(n-1) * c_1 + c_2$$

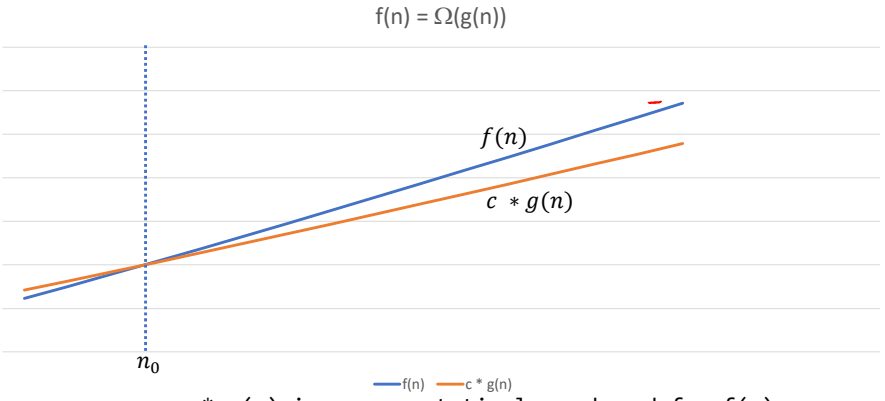
Big-Omega notation

Let $f(n)$ and $g(n)$ are increasing functions of n

$f(n)\ is\ \Omega\big(g(n)\big)$ (pronounced as big-Omega of)
if for some constants $c > 0$ and $n_0 \geq 1$

$$f(n) \geq cg(n), \qquad for\ all\ n > n_0$$

Big-Omega notation



c * g(n) is an asymptotic lower bound for f(n)

Big-Omega

$3nlogn + 2n$ $\Omega(n\log n)$ $n\log n \leq 3n\log n + 2n$ $c(1)$

$3n^3 + 2n^2 + 10$ $\Omega(n^3)$

1000 $\Omega(1)$

Big-Omega

$3nlogn + 2n \quad = \Omega(\text{nlogn})$

$3n^3 + 2n^2 + 10 \quad = \Omega(n^3)$

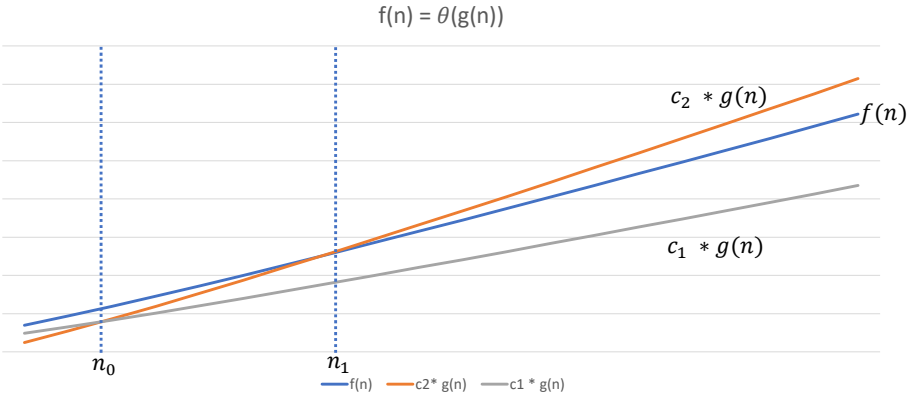
$1000 \quad = \Omega(1)$

Big-Theta notation

Let $f(n)$ and $g(n)$ are increasing functions of n

$f(n)\ is\ \theta\big(g(n)\big)$ (pronounced as big-Theta of)
if $f(n)\ is\ both\ O\big(g(n)\big)$ and $\Omega(g(n))$

Big-Theta notation



Big-Theta

$3nlogn + 2n$ $\Theta(n \log n)$

$3n^3 + 2n^2 + 10$ $\Theta(n^3)$

1000 $\Theta(1)$

Big-Theta

$3nlogn + 2n \quad = \theta(\text{nlogn})$

$3n^3 + 2n^2 + 10 \quad = \theta(n^3)$

$1000 \quad = \theta(1)$