Analysis of Algorithms

Analysis of Algorithms- Review

- ***Efficiency*** of an algorithm can be measured in terms of :
  - ***Time complexity***: a measure of the amount of time required to execute an algorithm
  - ***Space complexity***: amount of memory required
- Which measure is more important?
  - It often depends on the limitations of the technology available at time of analysis (e.g. processor speed vs memory space)

Time Complexity Analysis

- ***Objectives*** of time complexity analysis:
    - To determine the efficiency of an algorithm by computing an ***upper bound*** on the amount of work that it performs
    - To compare different algorithms before deciding which one to implement
- Time complexity analysis for an algorithm is ***independent*** of the programming language and the machine used

Time Complexity Analysis

- Time complexity expresses the relationship between
  - the ***size of the input***
  - and the ***execution time*** for the algorithm

Time Complexity Measurement

- Based on the number of ***basic or primitive operations*** in an algorithm:
  - Number of arithmetic operations performed
  - Number of comparisons
  - Number of Boolean operations performed
  - Number of array elements accessed
  - etc.
- Think of this as the ***work*** done

Example: Polynomial Evaluation

Consider the polynomial

$$P(x) = 4x^4 + 7x^3 - 2x^2 + 3x^1 + 6$$

Suppose that exponentiation is carried out using multiplications. Two ways to evaluate this polynomial are:

Brute force method:

P(x) = 4*x*x*x*x + 7*x*x*x – 2*x*x + 3*x + 6

Horner's method:

P(x) = (((4*x + 7) * x – 2) * x + 3) * x + 6

Method of analysis

- What are the ***basic operations*** here?
  - multiplication, addition, and subtraction
- We will look at the ***worst case*** (maximum number of operations) to get an ***upper bound*** on the work and thus of the running time of the algorithm

Method of analysis

General form of a polynomial of degree **n** is

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x^1 + a_0$$

where a_n is non-zero for all **n >= 0** (this is the worst case)

Analysis of Brute Force Method

$P(x) = a_n * x * x * \ldots * x * x +$

$a_{n-1} * x * x * \ldots * x * x +$

$a_{n-2} * x * x * \ldots * x * x +$

$\ldots +$

$a_2 * x * x +$

$a_1 * x +$

a_0

n multiplications

n-1 multiplications

n-2 multiplications

…

2 multiplications

1 multiplication

n total additions

Number of operations needed in the ***worst case*** is

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

$$= n\,(n + 1) / 2 \;+ n$$ ***(see below)***

$$= n^2 / 2 + 3n / 2$$

Sum of first n natural numbers:

Write the n terms of the sum in forward and reverse orders:

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

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

Add the corresponding terms:

$$2{*}t(n) = (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1) + (n+1)$$

$$= n\,(n+1)$$

Therefore, $t(n) = n\,(n+1) / 2$

Analysis of Horner's Method

$P(x) = (\ldots (((a_n * x +$	1 multiplication
$a_{n-1}) * x +$	1 multiplication
$a_{n-2}) * x +$	1 multiplication
$\ldots +$	
$a_2) * x +$	1 multiplication
$a_1) * x +$	1 multiplication
a_0	

n times

n total additions

Analysis of Horner's Method

Number of operations needed in the ***worst case*** is *:*

$$T(n) = n + n = 2n$$

Big-Oh Notation

- Analysis of Brute Force and Horner's methods came up with ***exact formulae*** for the maximum number of operations
- In general, though, we want to determine the running time, not the number of operations: Thus, we use the Big-Oh notation introduced earlier …

Big-Oh : Formal Definition

- ***Time complexity*** ***T(n)*** of an algorithm is ***O(f(n))*** (we say "***of the order f(n)***") if for some positive constant **c** and for all but finitely many values of **n** (**i.e.** as **n** gets large)

T(n) <= c * f(n)

- What does this mean? this gives an ***upper bound*** on the number of operations, ***for sufficiently large n***

Big-Oh Analysis

- We want the complexity function **f(n)** to be an easily recognized ***elementary function*** that describes the performance of the algorithm

Big-Oh Analysis

Example: Polynomial Evaluation

- What is the time complexity **f(n)** for Horner's method?
  - **T(n) = 2n**, so we say that the number of multiplications in Horner's method is **O(n)** ("*of the order of n*") and that ***the time complexity of Horner's method is O(n)***

Big-O Analysis

Example: Polynomial Evaluation

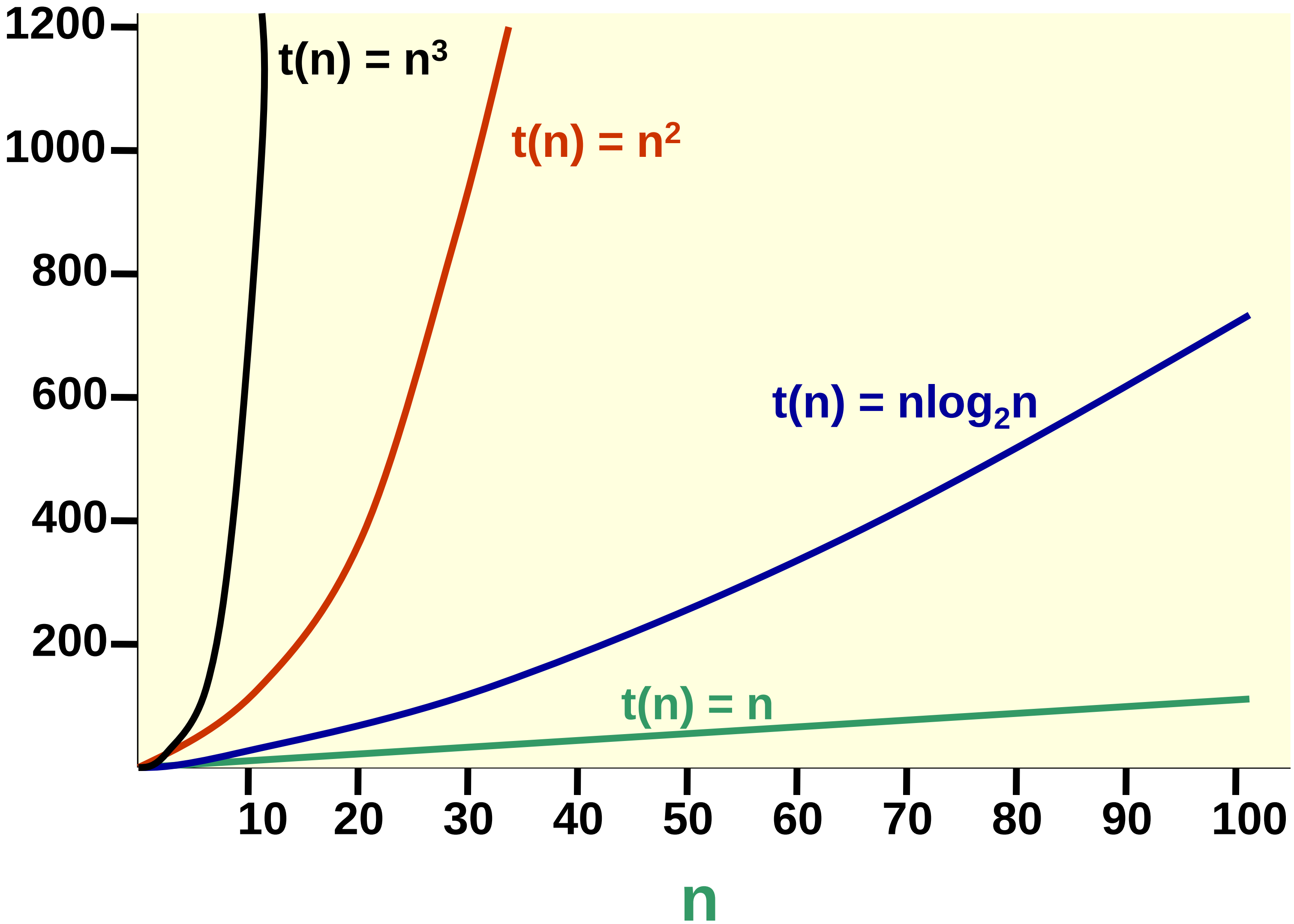
- What is the complexity **f(n)** for the Brute Force method?
  - Choose the highest order (***dominant***) term of

    $T(n) = n^2/2 + 3n/2$

    so

    $T(n)$ is $O(n^2)$

Recall: Shape of Some Typical Functions

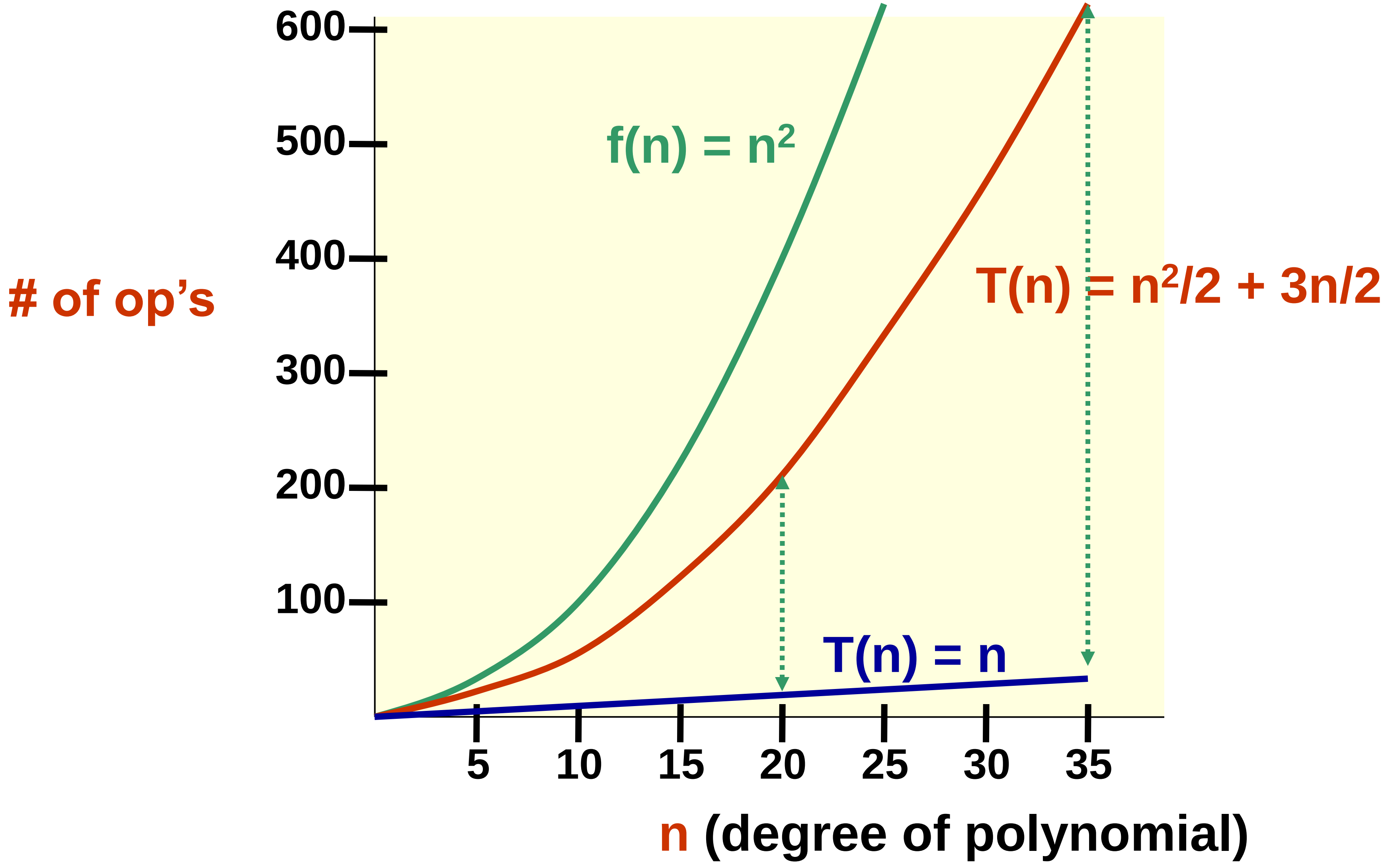


Big-Oh Example: Polynomial Evaluation Comparison

n	T(n) =2n (Horner)	T(n)= $n^2/2 + 3n/2$ (Brute Force)	f(n) = n^2
5	10	20	25
10	20	65	100
20	40	230	400
100	200	5150	10000
1000	2000	501500	1000000

n is the degree of the polynomial.

Big-Oh Example: Polynomial Evaluation



Time Complexity and Input

- Running time can depend on the ***size of the input*** (***e.g.*** sorting 5 items vs. 1000 items)
- Running time can also depend on the ***particular input*** (***e.g.*** suppose the input is already sorted)
- This leads to several kinds of time complexity analysis:
  - ***Worst case*** analysis
  - ***Average case*** analysis
  - ***Best case*** analysis

Worst, Average, Best Case

- ***Worst case analysis***: considers the ***maximum*** of the time over all inputs of size **n**
  - Used to find an upper bound on algorithm performance
- ***Average case analysis***: considers the ***average*** of the time over all inputs of size **n**
  - Determines the average (or expected) performance
- ***Best case analysis***: considers the ***minimum*** of the time over all inputs of size **n**

Discussion

- What are some difficulties with average case analysis?
  - Hard to determine
  - Depends on distribution of inputs (they might not be evenly distributed)
- So, we usually use ***worst case*** analysis (why not best case analysis?)

Example: Linear Search

- ***The problem***: search an array **A** of size **n** to determine whether it contains some value **key**
  - Return ***array index*** if found, ***-1*** if not found

Algorithm linearSearch (A, n, key)
In: Array A of size n and value key
Out: Array index of key, if key in A; -1 if key not in A

```
k = 0
while (k < n-1) and (A[k] != key) do
    k = k + 1
if A[k] = key then return k
else return –1.
```

- Total amount of work done:
  - ***Before loop***: a constant number c_1 of operations
  - ***Each time through loop***: a constant number c_2 of operations (comparisons, the **and** operation, addition, and assignment)
  - ***After loop***: a constant number c_3 of operations
- ***Worst case***: need to examine all **n** array locations, so the **while** loop iterates n times
- So, **$T(n) = c_1 + c_2 n + c_3$**, and the time complexity is **O(n)**

- ***Average*** case for a ***successful*** search:
  - Number of **while** loop iterations needed to find the key? **1** or **2** or **3** or **4** … or **n**
  - Assume that each possibility is equally likely
  - Average number of iterations performed by the **while** loop:

    (1+2+3+ … +n)/n = (n*(n+1)/2)/n

    = (n+1)/2
  - Average number of operations performed in the average case is $c_1 + c_3 + c_2(n+1)/2$. The time complexity is therefore **O(n)**

Example: Binary Search

- ***S**earch* a ***sorted*** array **A** of size **n** looking for the value **key**
- ***Divide and conquer*** approach:
  - Compute the middle index **mid** of the array
  - If **key** is found at **mid**, we are done
  - Otherwise repeat the approach on the half of the array that might still contain **key**

Binary Search Algorithm

Algorithm binarySearch (A,n,key)
In: Array A of size n and value key
Out: Array index of key, if key in A; -1 otherwise

```
first = 0
last = n-1
do {
   mid = (first + last) / 2
   if key < A[mid] then last = mid – 1
   else first = mid + 1
 } while (A[mid]  != key) and (first <= last)
 if A[mid] = key then return mid
 else return –1
```

- Number of operations performed before and after the loop is a constant c_1, and is independent of **n**

- Number of operations performed during a single execution of the loop is constant, c_2

- Time complexity depends on the number of times the loop is executed, so that is what we will analyze

Worst case**:** **key** is not found in the array

- Each time through the loop, at least half of the remaining locations are rejected:
  - After ***first*** time through, **<= n/2** remain
  - After ***second*** time through, **<= n/4** remain
  - After ***third*** time through, **<= n/8** remain
  - After ***k*** th time through, **<=** $n/2^k$ remain

- Suppose in the ***worst case*** that the maximum number of times through the loop is **k**; we must express **k** in terms of **n**
- Exit the **do..while** loop when the number of remaining possible locations is less than 1 (that is, when **first > last**): this means that **$n/2^k < 1$** and so **$n > 2^k$.**

Taking base-2 logarithms we get, $k < \log_2 n$.

Therefore, the total number of operations performed by the algorithm is at most

$c_1 + c_2 \log_2 n$ and so the time complexity is $O(\log_2 n)$ or just $O(\log n)$.

Big-Oh Analysis in General

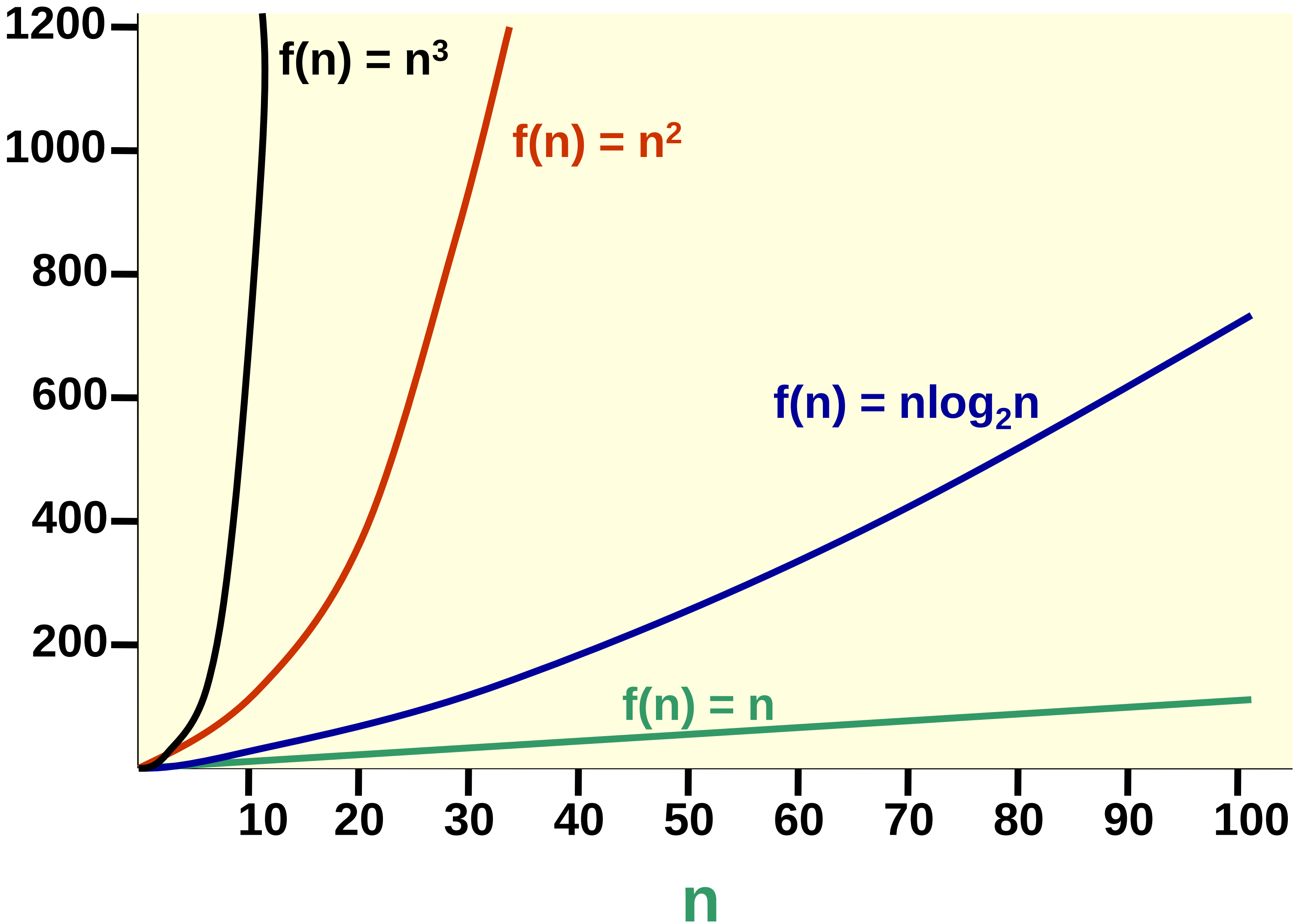
- To determine the time complexity of an algorithm:
  - Identify the basic operation(s)
  - Carefully analyze the most expensive parts of the algorithm: loops and calls
  - Express the number of operations as **$f_1(n) + f_2(n) + \ldots$**
  - Identify the ***dominant term*** **f_i**
  - Then the time complexity is **$O(f_i)$**

- ***Examples*** of ***dominant terms***:
  - $\mathbf{n}$ dominates $\mathbf{log_2(n)}$
  - $\mathbf{n\ log_2(n)}$ dominates $\mathbf{n}$
  - $\mathbf{n^2}$ dominates $\mathbf{n\ log_2(n)}$
  - $\mathbf{n^m}$ dominates $\mathbf{n^k}$ when $\mathbf{m > k}$
  - $\mathbf{a^n}$ dominates $\mathbf{n^m}$ for any $\mathbf{a > 1}$ and $\mathbf{m >= 0}$
- That is, for sufficiently large n,

$$log_2(n) < n < n\ log_2(n) < n^2 < \ldots < n^m < a^n$$

for $\mathbf{a > 1}$ and $\mathbf{m > 2}$

Recall: Shape of Some Typical Functions



Examples of Big-Oh Analysis

- ***Independent nested loops***:

```
int x = 0;
for (int i = 1; i <= n/2; i++){
   for (int j = 1; j <= n*n; j++){
         x = x + i + j;
   }
}
```

  - Number of iterations of inner loop is **independent** of the number of iterations of the outer loop (***i.e.*** the value of **i**)
  - How many times through outer loop?
  - How many times through inner loop?
  - Time complexity of algorithm?

- ***Dependent nested loops*:**

```
int x = 0;
for (int i = 1; i <= n; i++){
   for (int j = 1; j <= 3*i; j++){
         x = x + j;
   }
}
```

  - Number of iterations of inner loop ***depends on*** the value of **i** in the outer loop
  - On **i^{th}** iteration of outer loop, how many times through inner loop?
  - Total number of iterations of inner loop = sum for **i** running from **1** to **n**
  - Time complexity of algorithm?

Usefulness of Big-Oh

- We can ***compare algorithms*** for efficiency, for example:
  - ***Linear search*** vs ***binary search***
  - Different sort algorithms
  - Iterative vs recursive solutions (recall Fibonacci sequence!)

- We can ***estimate actual run times*** if we know the time complexity of the algorithm(s) we are analyzing

Estimating Run Times

- Assuming a million operations per second on a computer, here are some typical complexity functions and their associated runtimes:

f(n)	n = 10^3	n = 10^5	n = 10^6
$\log_2(n)$	10^{-5} sec.	$1.7*10^{-5}$ sec.	$2*10^{-5}$ sec.
n	10^{-3} sec.	0.1 sec.	1 sec.
$n \log_2(n)$	0.01 sec.	1.7 sec.	20 sec.
n^2	1 sec.	3 hours	12 days
n^3	17 mins.	32 years	317 centuries
2^n	10^{285} cent.	10^{10000} years	10^{100000} years

Discussion

- Suppose we want to perform a sort that is **$O(n^2)$**. What happens if the number of items to be sorted is 100000?
- Compare this to a sort that is **$O(n \log_2(n))$** . Now what can we expect?
- Is an **$O(n^3)$** algorithm practical for large **n**?
- What about an **$O(2^n)$** algorithm, even for small **n**? e.g. for a Pentium, runtimes are:

n = 30	n = 40	n = 50	n = 60
11 sec.	3 hours	130 days	365 years

Intractable Problems

- A problem is said to be ***intractable*** if solving it by computer is impractical
- Algorithms with time complexity $\mathbf{O(2^n)}$ take too long to solve even for moderate values of **n**
  - What are some examples we have seen?