WORKSHEET WEEK 01: ASYMPTOTIC BOUNDS

1.1 Introduction

1.1.1 Goals of this Course

Programming courses where we solve problems using language like C++ can have different focus.

- The programming language itself and its expressive power (for example, could have entire semester on various advanced features of C++).
- Object Oriented Programming and design. This could include creating the most appropriate OO design at the level of UML diagrams and Design patterns.
- Software engineering process – approaches to insure optimum quality and testability of the software.
- Efficiency of the algorithms being used – their time and space complexity.

The focus of this course is obviously the efficiency; namely – creating algorithms that can work on large input data sets or handle complex mathematical structures sufficiently fast. All the other aspects of programming serve the purpose of learning about algorithms and their efficiency.

- C++ language features being used – console applications (plaintext files for input and output). STL and C++ template classes are used to implement data structures.
- Object orientation serves to isolate concerns by separating data structure code from the remaining algorithms. Design patterns and other advanced OO stuff can be used, if students find it useful – but algorithm learning is largely independent from it.
- Ability to test our software (including unit tests) and also to measure the speed of our code is essential.

1.1.2 Formal Requirements

The maximum overall grade is 100% (100 percentage points). Your numeric grade will be obtained by dividing your actual percentage points by 10 and by rounding to the nearest integer. For example, the minimum number of percentage points to get the grade 10 is 95% (and the minimum number of percentage points to get the grade 4 is 35%).

All the percentage points in this section are relative to the overall maximum grade.

1. Four programming tasks (40% total; 10% per task). These problems ask you to solve some problem using algorithms and data structures covered in the class. (The first tasks let you use any data structures you want, but tasks #3 and #4 may ask you to refrain from using the built-in STL libraries and similar data structure implementations and to use your own.)

2. One design and solution description (20% total): Algorithm analysis, class design, test creation and finally – test-driven development. Half of the credit – 10 percentage points are given for the submitted analysis paper. Another 10 percentage points are given for the implementation.
3. Midterm (20%) – algorithmic tasks done on paper (writing pseudocode, drawing pictures of data structures and running some algorithmic steps there).
4. Final (20%) – similar to the midterm, but refers to the second half of the course.

Midterm and final will not ask you to implement any code. To prepare for these exams we will solve problems (especially on Wednesday 14:30 lab sessions) that resemble midterms and finals. Such training tasks will not affect your grade, but you are encouraged to practice them.

1.1.3 Office Hours

Use a link to register – https://calendly.com/kalvis-apsitis/office-hours?month=2022-02; instructor will send a Zoom link. You are encouraged to come in groups as well.

There will be strongly suggested office hours (about one 30 minute session) **before** the first programming task is due. This would help to ensure that your directory layout, compilation and testing approach is the same as that used by the instructor.

1.2 How to Determine Algorithm Efficiency?

1.2.1 Looking Up an Item in a List

Problem: There is a dictionary (the traditional printed book variety) with 50 000 words written in the English alphabet (26 letters; assume that all of them are lowercase). The words in the dictionary are sorted alphabetically. The task is to find a given word w (such as w = `efficiency`) in this dictionary or to report that there is no such word.

Note: There is also a data structure named *dictionary* (storing keys and values); here we use the everyday notion of a dictionary – an alphabetically arranged list of words.

Linear (Brute Force) Solution:

Let us have a zero-based dictionary D with n items from $D[0]$ to $D[n-1]$.

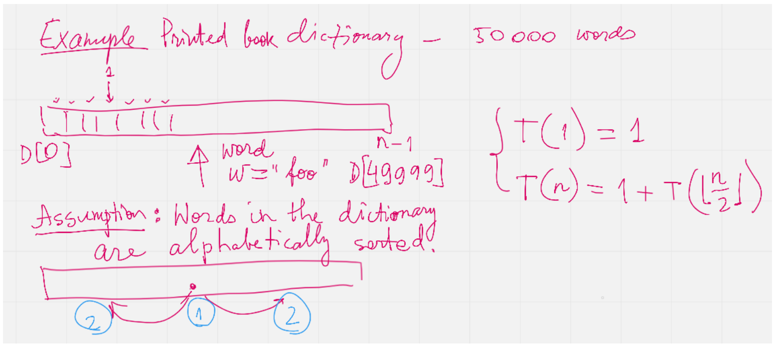
```
LINEARSEARCH($D, w$)
1.  for $i$ in RANGE($0, n$):
2.      if $w$ == $D[i]$:
3.          return FOUND $w$ at location $i$
4.  return NOT FOUND
```

This method in the real life would mean somebody scanning through all the words in a dictionary and searching for the match with the given word w. This would be very inefficient. The only advantage for this approach – we do not need any assumptions about the word order in the dictionary – the algorithm would work equally well even for totally unordered list of words.

Binary Search Solution:

Binary search is a different algorithm that relies on the alphabetical sorting of the dictionary $D[0] < D[1] < \ldots < D[n-2] < D[n-1]$. At every point we maintain a closed interval $[\ell, r]$ so that the index $D[i] = w$ we want to find satisfies the inequalities $\ell \leq i \leq r$.

The initial call is BINARYSEARCH(D, ℓ, r, w), where $\ell = 0$ and $r = n-1$. After that the binary search may call itself recursively on shorter intervals.


Example Printed book dictionary – 50 000 words
1
D[0]
n-1
word
w = "foo"
D[49999]
$T(1) = 1$
$T(n) = 1 + T(\lfloor \frac{n}{2} \rfloor)$
Assumption: Words in the dictionary are alphabetically sorted!
2
1
2

```
BINARYSEARCH$(D, \ell, r, w)$
1.  if $\ell > r$:
2.      return NOT FOUND $w$
3.  $m = \left\lfloor \frac{\ell + r}{2} \right\rfloor$
4.  if $w == D[m]$:
5.      return FOUND $w$ at location $m$
6.  else if $w < D[m]$:
7.      return BINARYSEARCH$(D, \ell, m-1, w)$
8.  else:
9.      return BINARYSEARCH$(D, m+1, r, w)$
```

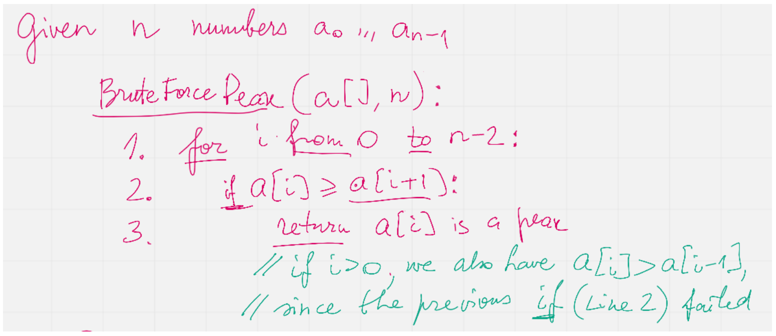
1.2.2 Finding a Peak in a Numeric Sequence

Definition: Given a sequence a_i ($i = 0, \ldots, n-1$) we call its element a_i a *peak* iff it is a local maximum (not smaller than any of its neighbors):

$$a_i \geq a_{i-1} \ \textbf{ and } \ a_i \geq a_{i+1}$$

In case if $i = 0$ or $i = n-1$, one of these neighbors does not exist; and in such cases we only compare a_i with neighbors that do exist.

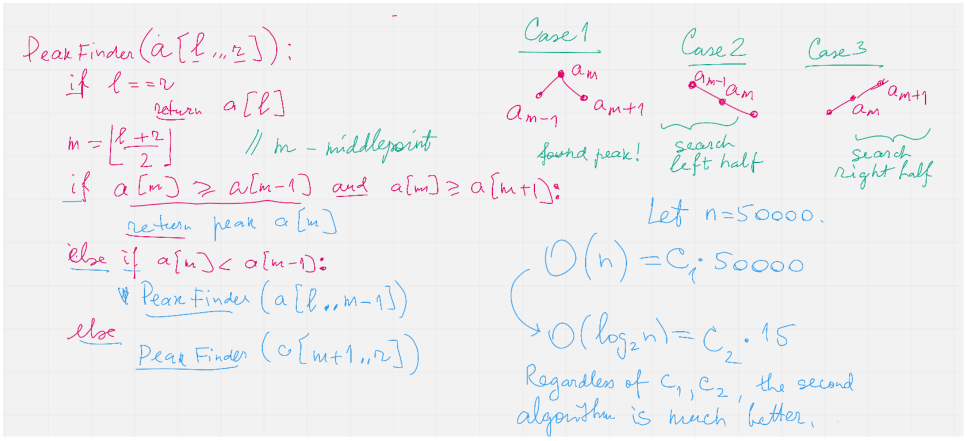
Brute Force Algorithm:



Given n numbers a_0 ,,, a_{n-1}

```
BruteForcePeak(a[], n):
1.  for i from 0 to n-2:
2.      if $a[i] \geq a[i+1]$:
3.          return a[i] is a peak
            // if i>0, we also have $a[i] > a[i-1]$,
            // since the previous if (Line 2) failed
```

Note: Observe that in every nonempty numeric sequence a_i there exist at least one peak (for example, the global maximum is always a peak). On the other hand, peaks are not necessarily unique. In particular, if the sequence is constant (all members are equal), then any member there is a peak.



```
PeakFinder(a[l ,,, r]):
    if l == r
            return a[l]
    m = ⌊(l+r)/2⌋          // m - middlepoint
    if a[m] ≥ a[m-1] and a[m] ≥ a[m+1]:
        return peak a[m]
    else if a[m] < a[m-1]:
        PeakFinder(a[l ,, m-1])
    else
        PeakFinder(a[m+1 ,, r])
```

Let $n=50000$.

$O(n) = C_1 \cdot 50000$

$O(\log_2 n) = C_2 \cdot 15$

Regardless of C_1, C_2, the second algorithm is much better.

1.2.3 Big-O-Notation

Definition: Let :math:g colon mathbb{N} rightarrow mathbb{R}_{0+}` be a function from natural numbers (non-negative integers) to non-negative real numbers. Then $O(g)$ is the set of all functions $f\colon \mathbb{N} \to \mathbb{R}$ such that there exist real constants $c > 0$ and $n_0 \in \mathbb{N}$ such that

$$\forall n \in \mathbb{N} \left(n \geq n_0 \to |f(n)| \leq c \cdot g(n) \right).$$

Examples: Show using the above definition of $O(g)$ the following facts:

(A) $f(n) = 13n + 7$ is in $O(n)$. (Formally, $f \in O(g)$, where $g(n) = n$.)

(B) $f(n) = 3n^2 - 100n + 6$ is in $O(n^2)$.

(C) $f(n) = 3n^2 - 100n + 6$ is in $O(n^3)$.

(D) $f(n) = 3n^2 - 100n + 6$ is **not** in $O(n)$.

For every example (every pair of functions $g(n)$ and $f(n)$) either find $c > 0$ and $n_0 \in \mathbb{N}$ such that the definition is satisfied, or demonstrate that such $c > 0$ and n_0 cannot exist.

Here are the definitions of three asymptotic concepts.

Definition: Let $g\colon \mathbb{N} \to \mathbb{R}_{0+}$ be a function from natural numbers (non-negative integers) to non-negative real numbers. Then $O(g)$ is the set of all functions $f\colon \mathbb{N} \to \mathbb{R}$ such that there exist real constants $c > 0$ and $n_0 \in \mathbb{N}$ satisfying $\forall n \in \mathbb{N} \left(n \geq n_0 \to |f(n)| \leq c \cdot g(n) \right)$.

Definition: Let $g\colon \mathbb{N} \to \mathbb{R}_{0+}$ be a function. Then $\Omega(g)$ is the set of all functions $f\colon \mathbb{N} \to \mathbb{R}$ such that there exist real constants $c > 0$ and $n_0 \in \mathbb{N}$ satisfying $\forall n \in \mathbb{N} \left(n \geq n_0 \to |f(n)| \geq c \cdot g(n) \right)$.

Definition: Define $\Theta(g)$ to be the intersection of $O(g)$ and $\Omega(g)$.

Formally, let $g\colon \mathbb{N} \to \mathbb{R}_{0+}$ be a function. Then $\Theta(g)$ is the set of all functions $f : \mathbb{N} \to \mathbb{R}$ such that there exist real constants $c_1, c_2 > 0$ and $n_0 \in \mathbb{N}$ satisfying

$$\forall n \in \mathbb{N} \left(n \geq n_0 \to c_1 \cdot g(n) \leq |f(n)| \leq c_2 \cdot g(n) \right).$$

In spoken language we often use descriptive concepts:

- If $f \in O(g)$, then $g(n)$ is called *asymptotic upper bound* of $f(n)$.
- If $f \in \Omega(g)$, then $g(n)$ is called *asymptotic lower bound* of $f(n)$.
- If $f \in \Theta(g)$, then $g(n)$ is called *asymptotic growth order* of $f(n)$.

Definition: The *time complexity* of an algorithm is described by a function $f(n)$, if for **any** input of length n bytes, the time spent running the algorithm is bound from above by $f(n)$ (starting from some natural n_0).

Intuition about these definitions:

1. Why do we need asymptotic behavior (namely, $\forall n \geq n_0$)? What about small values n?

   Asymptotic behavior ignores $n < n_0$ for some n_0, since algorithmic complexity on short inputs does not matter very much. Theoretically, you could even cheat – remember a large lookup table containing all sorts of inputs of length $n < n_0$ (with precomputed correct answers). Clearly, this does not tell us anything about the performance of this algorithm – algorithms differ on how they behave on long inputs.

   Another reason is the simplicity of the functional behavior as $n \to \infty$. Even though we would love to predict the speed of an algorithm for short input lengths n, the dependence on n is likely quite complex (and we cannot ignore lower order terms). As n becomes very large, only the dominant parts in the expression $f(n)$ matter.

2. Why do we allow arbitrary constant c in the inequalities like $|f(n)| \leq c \cdot g(n)$?

   Trying to measure computation costs with explicit constants would make the cost model more complicated – it would matter how many CPU commands and machine-words are involved. Also, the meaning of constants change as soon as you obtain a faster computer – a constant speedup is not hard to achieve.

1.3 Properties of Big-O, Big-Omega, Big-Theta

Big-O and Limit of the Ratio: If the following limit exists and is finite:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C < +\infty,$$

then $f(n)$ is in $O(g(n))$.

Big-O isătransitive: Ifă$f(n) \in O(g(n))$ andă$g(n) \in O(h(n))$, thenă$f(n) \in O(h(n))$.

Sum of two functions: Ifă$f(n) \in O(h(n))$ andă$g(n) \in O(h(n))$, then $f(n) + g(n) = O(h(n))$.

All polynomials: Anyăk-thădegree polynomial $P(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ is in $O(n^k)$.

Logarithms of any base: If $a, b > 1$ are any real numbers, then $\log_a n = O(log_b n)$. Typically use just one base (usually, it is base 2 or base e of the natural logarithm, if you prefer that), and write just $O(\log n)$ without specifying base at all.

The last result directly follows from the formula to change the base of a logarithm: $\forall a, b, m > 1 \left(\log_a b = \frac{\log_m b}{\log_m a} \right)$.

1.4 Examples with Big-O, Big-Omega, Big-Theta

1.4.1 The Complexity of Combined Algorithms

Once the the complexity of constituent parts of an algorithm is known, these complexities can be combined to find the time complexity of the overall algorithm.

Example1: ALGORITHMA(INPUT) has time complexity in $O(n^a)$, but ALGORITHMB(INPUT) has time complexity in $O(n^b)$. What is the complexity of $AlgorithmC$ that first calls ALGORITHMA(INPUT), then calls ALGORITHMB(INPUT), and finally somehow combines the results in $O(1)$ time.

```
ALGORITHMC(input)
    resultA = ALGORITHMA(input)
    resultB = ALGORITHMB(input)
    return COMBINE(resultA, resultB)
```

Example2: ALGORITHMA(INPUT) has time complexity in $O(n^a)$, but ALGORITHMB(INPUT) has time complexity in $O(n^b)$. What is the complexity of $AlgorithmD$ that first calls ALGORITHMA(INPUT), then calls ALGORITHMB(INPUT) in a long loop n times and returns the Boolean conjunction of all the results.

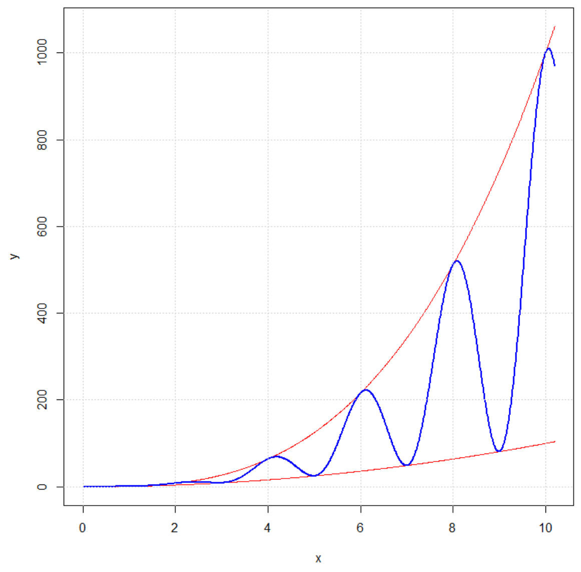
```
ALGORITHMD(input)
    result = ALGORITHMA(input)
    for $i$ from $1$ to $n$:
        result = result $\wedge$ ALGORITHMB(input)
    return result
```

Example3: Define a function $f\colon \mathbb{N} \rightarrow \mathbb{R}_{0+}$ which infinitely often takes both values n^2 and also n^3.

Solution: One could define a function with a condition (n^2 for even n and n^3 for odd n):

$$f(n) = \begin{cases} n^3, & \text{if } n \text{ is even} \\ n^2, & \text{if } n \text{ is odd} \end{cases}$$

This function with two branches is defined just for natural numbers $n \in \mathbb{N}$ (which is enough for Big-O notation concept). If you wish, this function can also be defined for all real numbers $x \in \mathbb{R}$: $f(x) = x^2 + \frac{1}{2}\left(1 + cos(\pi x)\right)\left(x^3 - x^2\right)$.



Note: This example shows a function f which is in $O(n^3)$ and also in $\Omega(n^2)$, but it does not belong to any $\Theta(g)$ for some simple function g (except to its own class $f \in \Theta(f)$). One could come up with an algorithmic task which is considerably faster for odd-length inputs. And also vice versa: Some algorithmic task may be faster for even-length inputs. Hence, there is no total order among the asymptotic growth rates – sometimes asymptotic growth rates are incomparable.

1.4.2 Slowly growing functions

Example4: Show that $f(n) = 0$ is not in $\Omega(1)$.

Solution: Apply the definition of $f \in \Omega(g)$, where $f(n) = 0$, but $g(n) = 1$. The required inequality $|f(n)| \geq c \cdot g(n)$ which is, in fact, $0 \geq c \cdot 1$ is never true, if $c > 0$.

Let us have a less trivial example – a strictly positive function which does not have $g(n) = 1$ as its asymptotic lower bound. In fact, any infinite sequence having a subsequence converging to 0 is fine.

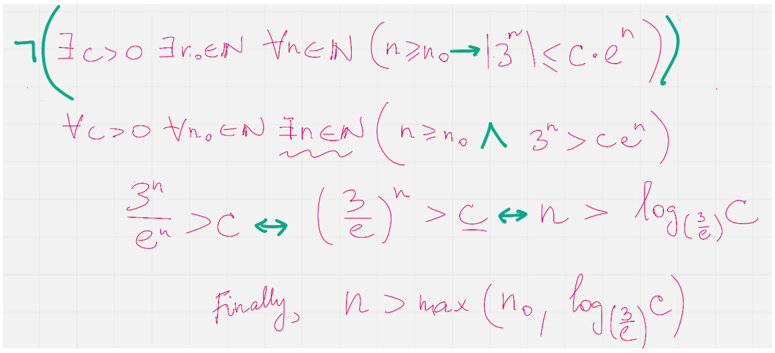
Example5: Show that $f(n) = \frac{1}{n}$ is not in $\Omega(1)$.

Solution: Let us pick some $c > 0$ and some n_0 first. We have to show that there must exist a number $n > n_0$ that violates the inequality from the definition of $f \in O(g)$.

Let us pick n such that $n > n_0$ and $n > 1/c$. Then $|f(n)| = 1/n < c \cdot 1$, which means that the required inequality $|f(n)| = 1/n \geq c \cdot 1$ does not hold.

Example6: Use the Big-O definition to show that $f(n) = 3^n$ is not in $O(e^n)$.

Hint for Example3: Let us use the formal definition of Big-O notation, and show that its negation is true.



$$\neg\left(\exists c>0\ \exists n_0\in\mathbb{N}\ \forall n\in\mathbb{N}\ \left(n\geq n_0 \to |3^n|\leq c\cdot e^n\right)\right)$$

$$\forall c>0\ \forall n_0\in\mathbb{N}\ \underset{\sim\sim}{\exists n\in\mathbb{N}}\ \left(n\geq n_0 \wedge 3^n > ce^n\right)$$

$$\frac{3^n}{e^n} > C \leftrightarrow \left(\frac{3}{e}\right)^n > \underline{C} \leftrightarrow n > \log_{(\frac{3}{e})} C$$

Finally, $n > \max\left(n_0, \log_{(\frac{3}{e})} C\right)$

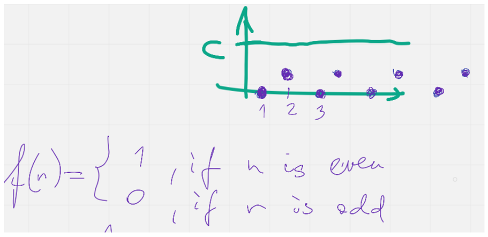
We see that picking sufficiently large n makes the inequality from the Big-O definition false. Meanwhile $e^n \in O(3^n)$, since 3^n is always larger; so one can pick $n_0 = 0$ and $c = 1$.

Note: We observe that $f(n) = e^n$ is in $O(3^n)$, but 3^n is not in $O(e^n)$. Unlike logarithms (which only differ by a constant factor – and are all in the Big-O relation with each other), any two different bases for exponent functions (such as $e \approx 2.71$ and 3) create different asymptotic growth rates.

Example7: If a function $f(n) = C$ is constant, then it is $O(1)$. Is the converse also true – does the statement $f \in O(1)$ imply that f is a constant function.

Hint for Example4: Can define function by cases so that it is *bounded* (see https://bit.ly/3Bdv1aR), but not equal to the (same) constant.

Solution: The function that is not constant, but still in $O(1)$ is shown in the image below.

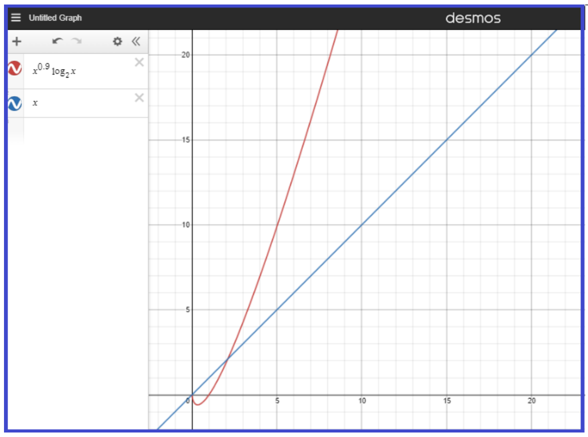


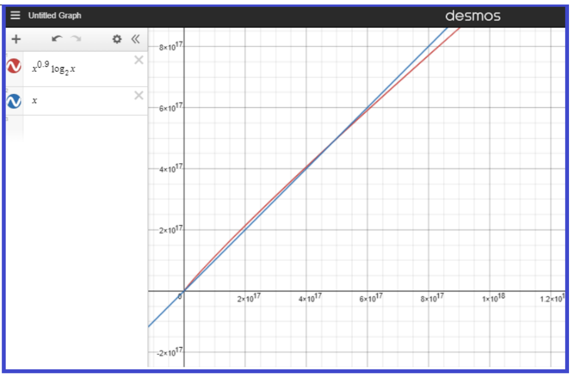
$$f(n) = \begin{cases} 1, & \text{if } n \text{ is even} \\ 0, & \text{if } n \text{ is odd} \end{cases}$$

Example8: Define functions $g(n) = n$ and $h(n) = n^{0.9} \cdot \log_2 n$ Just as in the above example we can show that the limit $h(n)/g(n) = 0$ as $n \to \infty$.

Draw graphs of the functions $f_1(x) = x$ and $f_2(x) = x^{0.9} \cdot \log_2 x$ and observe, for what values n $f_2(n)$

This example shows that establishing the Big-O properties using a graphing calculator could be difficult and misleading – sometimes the asymptotic behavior becomes evident only for huge values of n_0.





1.4.3 Fast Growing Functions and Slow Algorithms

Definition: Binomial coefficients show in how many ways an unordered selection of k elements out of n elements can be made:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}.$$

Certain algorithms rely on trying out all possible combinations of some data. In such cases the amount of work may grow exponentially in terms of the input length n. One such problem is *Traveling Salesman* – currently there is no known efficient algorithm for this problem.

Among the many functions that grow very fast (and are time complexities of algorithms that are very slow) some are much faster than the others. In particular, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then function $g(n)$ grows much faster than $f(n)$ and $f \in O(g)$, but $g \notin O(f)$

Example9: Order these functions in increasing order with respect to their Big-O growth rate:

- $f_1(n) = 2^{2^{10000}}$
- $f_2(n) = 2^{10000n}$
- $f_3(n) = \binom{n}{2} = C_n^2$
- $f_4(n) = \binom{n}{\lfloor n/2 \rfloor}$
- $f_5(n) = \binom{n}{n-2}$
- $f_6(n) = n!$
- $f_7(n) = n\sqrt{n}$

Solution: Eliminate a few functions which do not exceed polynomials (polynomial-time algorithms are not considered exceptionally slow). f_1 is just $O(1)$, f_7 is in $O(n^{1.5})$, but functions $f_3(n) = f_5(n) = \frac{n(n-1)}{2}$ which is in $O(n^2)$.

It remains to order the remaining functions (all of them grow fast - they are exponential in terms of n). We will prove that their order is the following: f_4, f_2, f_6 – see the two following examples.

Lemma: We have the following estimate:

$$\binom{n}{\lfloor n/2 \rfloor} \sim \frac{4^n}{\sqrt{\pi n}}$$

To prove this, apply Stirlings formula: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

Note: Very similar expression describes Catalan numbers. The n-th Catalan number is defined by the following equality: $C_n = \frac{1}{n+1}\binom{2n}{n}$. They arise in various combinatorial problems. See https://bit.ly/3stVNIk for details. (For example, some computer algorithm that would process every valid way how to parenthesize an expression consisting of n terms will require C_n steps.)

Example10: Show that $f_4(n) = \binom{n}{\lfloor n/2 \rfloor}$ is in $O(f_2)$ where $f_2(n) = 2^{10000n}$.

Solution: From the Lemma we immediately see that $f_4(n)$ is in $O(n^4)$. And in turn n^4 is in $O(2^{10000n})$. Note that $2^{10000n} = (2^{10000})^n$ – it is also an exponential function, but the exponent base 2^{10000} is larger than 4.

Lemma: Let $g(n) = n!$ and $f(n) = a^n$ for some constant a. Then $f \in O(n!)$ and also $\lim_{n\to\infty} \frac{a^n}{n!} = 0$. In other words, factorial grows faster than any exponential function.

Proof: Define the constant $N = 2a$. Denote $\frac{a^N}{N!} = C$. Initially set $n = N$. Every time you increment n, the numerator increases exactly a times, but denominator increases at least $N = 2a$ times. Therefore every time you increment n to $n + 1$ the fraction will decrease at least twice. The only number that can be a limit of such a sequence is 0.

Example11: Show that $f_2(n) = 2^{10000n}$ is in $O(f_6)$ where $f_6(n) = n!$.

Solution: This immediately follows from the previous Lemma, where $a = 2^{10000}$. The values of n for which $n!$ grows faster than 2^{10000n} are very large; they start at 2^{10000}.