DET102 Data Structures and Algorithms

Lecture 01: Course Introduction and Algorithm complexity

Course Information

- Time: 9:00-12:00 Friday
- Avenue: 802 in CBCC

- Dual-mode is provided.
  - F2F: full-time students
  - Microsoft Teams: Part-time students

- Video will be provided for self-learning and review.
- Online exercise on Moodle will be used to take attendance.

- Lecturer
  - Yingchao ZHAO (趙英超)
  - Office: A805
  - Tel: 3702 4206
  - Email: yczhao@cihe.edu.hk

- Consultation Hour:
  - Thursday 13:00-17:00
- Online Q&A (optional)
  - Wednesday 21:00-22:00
  - On teams.

Assignments & Grading

- **Online Exercise:**
  - Available in moodle for a week. 20% of the final grade
  - You can try twice, and we take the highest mark.
  - We also use it as attendance. Your attendance rate should be at least 80%.
- **Programming Project:**
  - Two Individual projects: P1 and P2
  - Must be done in C/C++/Python.
  - Project (demo+report): 40% of the final grade.
- **Final Exams:**
  - Paper based : 40% of final grade

Assignments & Grading

- **Academic Honesty:**
  - All classwork should be done independently, unless explicitly stated otherwise on the assignment handout.
  - You may discuss general solution strategies, but you must write up the solutions yourself.
  - You may refer to materials from internet, but you must cite those materials in your submission.
- NO LATE SUBMISSION ACCEPTED
  - Turn in what you have at the time it's due.
  - All exercises are due at the start of next class.
  - Late projects will be accepted, but you will be penalized.

Resources

- Online Resources
  - Dave Mount's Lecture Notes: http://www.cs.umd.edu/~mount/420/Lects/420lects.pdf
  - Stanford C programming: https://www.youtube.com/playlist?list=PLD28639E2FFC4B86A
  - Stanford CS Education Library: http://cslibrary.stanford.edu/
  - Khan Academy: https://www.khanacademy.org/computing/computer-science/algorithms#concept-intro
- Books
  - Mark, A. Weiss (2013). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.
  - Agarwal, B. and Baka, B. (2018). *Hands-On Data Structures and Algorithms with Python: Write complex and powerful code using the latest features of Python 3.7* (2nd ed.). Packt Publishing
  - Cormen, T. H., Leiserson, C. E, Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). Cambridge, Mass.: MIT Press.

Online Judgement

1. Aizu Online Judge (AOJ)
   - http://judge.u-aizu.ac.jp
2. Hangzhou Dianzi University Online Judge (HDU)
   - https://acm.hdu.edu.cn/
3. UVa Online Judge (UVa)
   - https://onlinejudge.org/
4. Leetcode
   - https://www.leetcode.com/

Programming Competition

1. Baidu BestCoder
   - http://bestcoder.hdu.edu.cn/contests/contest_list.php
2. ACM International Collegiate Programming Contest
   - https://icpc.global/
3. CodeForces
   - https://codeforces.com/

What is Algorithm ?

- A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.
- Example

Algorithm in computer science

- A sequence of elementary computational steps that transform the input into the output.
- A tool for solving well-specified computational problems, e.g., Sorting, Matrix Multiplication
- What do we need to do with an algorithm?
  - Correctness Proof:

    for every input instance, it halts with the correct output
  - Performance Analysis (1 second or 10 years, 1K or 10G):

    How does the algorithm behave as the problem size gets large

    both in running time and storage requirement

Algorithm Example: Top 3

- You are given **10** scores. You need to print the top **3** scores in non-increasing order. Note: the scores are integers between 0 and 100.

- Sample input:

  25 36 4 55 71 18 0 71 89 65

- Sample output:

  89 71 71

Algorithm 1: Search three times

1. Save all the scores in an array A[10]
2. Find the largest number in A and output it
3. Remove the number found in step 2 from A. Find the largest number in the remaining 9 numbers and output it.
4. Remove the number found in step 3 from A. Find the largest number in the remaining 8 numbers and output it.

Algorithm 2: Sorting first

1. Save all the scores in an array A[10]
2. Sort A in decreasing (non-increasing) order.
3. Output the first three elements of A in order.

Algorithm 3: Counting frequency

1. Count the frequency of number p and save it in array C[p]
2. Check array C in the order of C[100], C[99], C[98],….

   If C[p]>0, output p for C[p] time(s) until we totally output 3 numbers.

How to describe an algorithm?

Pseudocode

- Pseudocode is an artificial and informal language that helps programmers develop algorithms.
- Pseudocode is a "text-based" detail (algorithmic) design tool.
- Pseudocode often uses structural conventions of a normal programming language, but it is intended for human reading rather than machine reading.
- It typically omits details that are essential for machine understanding of the algorithm, such as variable declarations and language-specific code.

Pseudocode

- Denote variable in English.
- Omit variable declaration and type
- Use if, while, for,…
- Use indentation rather than { } to represent blocks
- =, ==, !=, | |, &&, !
- A[i] represents the i-th item in array A.

Pseudo code example

```
// Algorithm 1: search three times

for i from 1 to 10
      A[i]=the i-th score
a= max value in A
remove a from A
b= max value in A
remove b from A
c= max value in A
output a,b,c
```

Task 1:
Try to write pseudocode for Algorithm 2 and Algorithm 3

Generalized problem: Top n

- You are given **m** scores a_i (i=1,2,…,m). You need to print the top **n** scores in non-increasing order.
- Constraints:
  - m<=1000000
  - n<=1000
  - 0<=a_i<=1000000

Which algorithm to use ?
- concise
- easy to code
- efficiency
- memory used

Algorithm Complexity

- How to measure efficiency of an algorithm ?
  - Time complexity: how much is used in CPU/GPU ?
  - Space complexity: how much is used in memory ?
- Balance between time complexity and space complexity.
- Time complexity usually causes more troubles.

Kinds of Analysis

(Usually) Worst case Analysis:

- T(n) = **max** time on **any** input of size n
- Knowing it gives us a guarantee about the upper bound.
- In some cases, worst case occurs fairly often

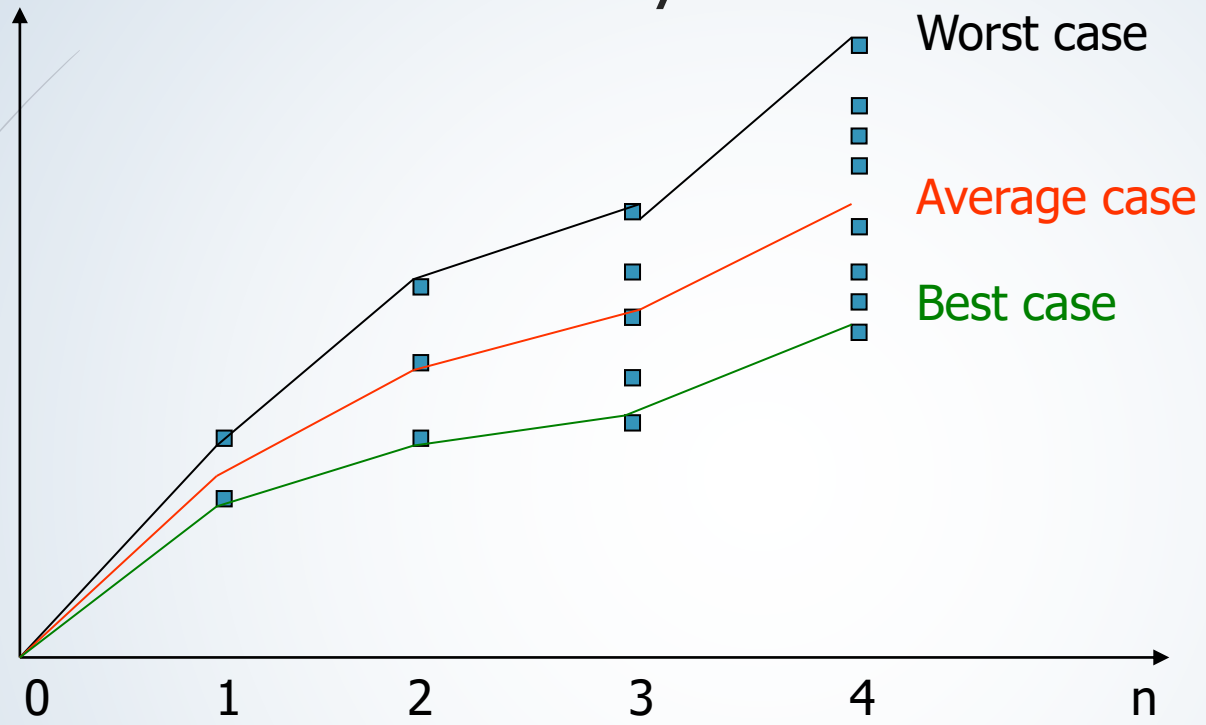
(Sometimes) Average case Analysis:

- T(n) = **average** time over **all** inputs of size n
- Average case is often as bad as worst case.

(Rarely) Best case Analysis:

- Cheat with slow algorithm that works fast on some input.
- Good only for showing bad lower bound.

Kinds of Analysis



- Worst Case: maximum value
- Average Case: average value
- Best Case: minimum value

Analyze algorithms

- While analyzing a particular algorithm, we usually count *the number of operations performed by the algorithm*.
- We focus on the number of operations, not on the actual computer time to execute the algorithm.
- This is because a particular algorithm can be implemented on a variety of computers and the speed of the computer can affect the execution time. However, the number of operations performed by the algorithm would be the same on each computer.

Primitive Operations

- Primitive operations are basic computations performed by an algorithm. Examples are *evaluating an expression*, *assigning a value to a variable*, *indexing into an array*, *calling a method*, *returning from a method*, etc. They are easily identifiable in pseudocode and largely independent from the programming language.
- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm as a function of the input size. Think about the worst case, best case, and average case.

Example

Algorithm arrayMax(A, n):	
Input: An array A of n integers	Number of operations:
Output: the max element	
max = A[0]	2
for i = 1 to n-1 do	n
if (A[i] > max) then max = A[i]	6(n-1) (including increment counter)
return max	1
	Total: 7n-3

Primitive Operations

The algorithm arrayMax executes about $7n - 3$ primitive operations in the worst case. Define:

- a = Time taken by the fastest primitive operation
- b = Time taken by the slowest primitive operation

Let $T(n)$ be the worst case time of arrayMax.
Then $a(7n - 3) <= T(n) <= b(7n - 3)$

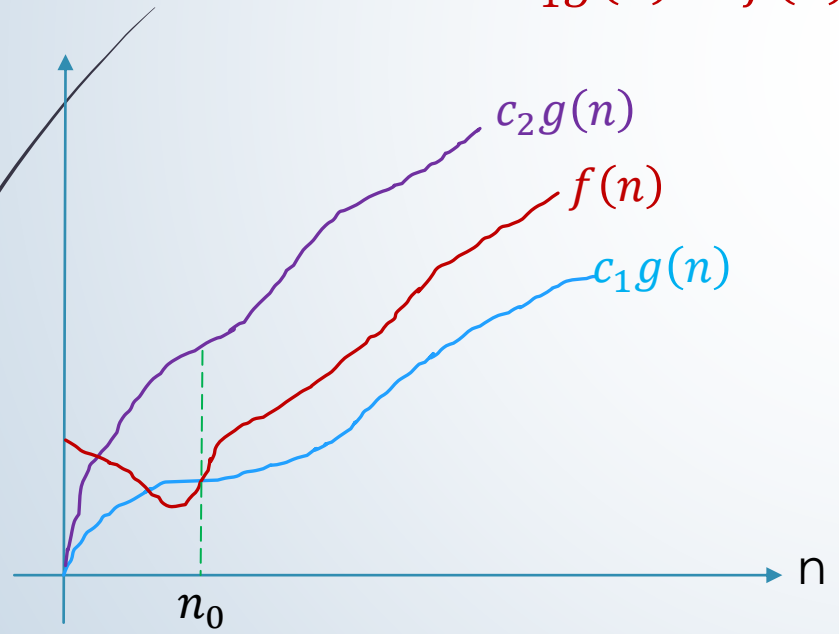
- The running time $T(n)$ is bounded by two linear functions
- Changing the hardware/software environment will not affect the *growth rate* of T(n)

Asymptotic Notation

- Asymptotic Tight Bound: Θ Intuitively like "=" (The Theta notation)
- Asymptotic Upper Bound: O Intuitively like "$\leqslant$" **(The Big-Oh notation)**
- Asymptotic Lower Bound Ω Intuitively like "≥" (The Little-Omega notation)
- Asymptotic Upper Bound: o Intuitively like "<" **(The Little-Oh notation)**
- Asymptotic Lower Bound: ω Intuitively like ">" (The Little-Omega notation)

Θ-notation

- For a given function g(n), we denote by Θ(g(n)) the set of functions
- Θ(g(n))={f(n): there exist positive constants c_1, c_2 and n_0 such that
- $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all n $\ge n_0$}



$f(n)$= Θ(g(n))

The value of f(n) always lies between $c_1 g(n)$ *and* $c_2 g(n)$ inclusively.

g(n) is an *asymptotically tight bound* for f(n)

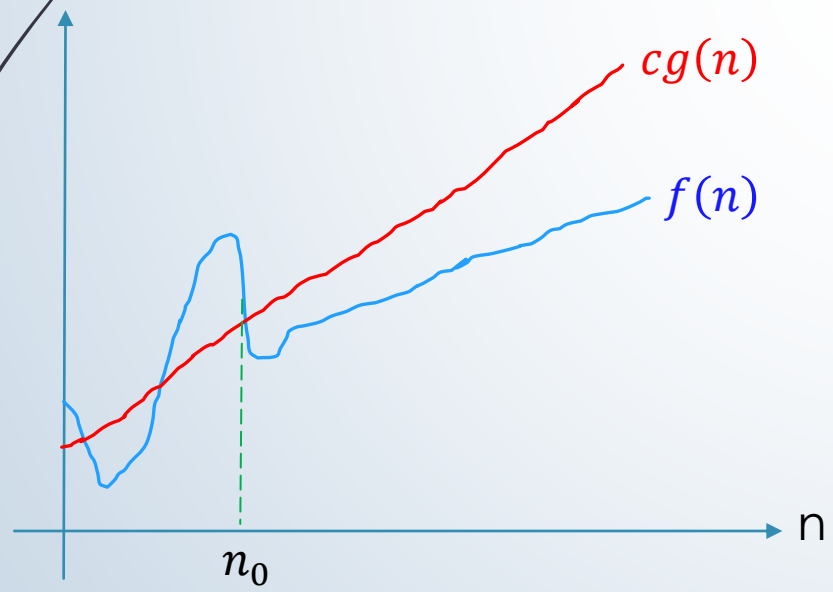
Θ-notation

- Example: To show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

- Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$ or $\Theta(1)$.
- $\Theta(1)$ means either a constant or a constant function with respect to some variable.

O-notation

- For a given function g(n), we denote by O(g(n)) the set of functions
- O(g(n))={f(n): there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n $\geq n_0$}



$f(n)$= O(g(n))

O-notation

- O(g(n))={f(n): there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

- Once the input size n gets big enough (bigger than n_0), then $f(n)$ is always less than some constant multiple of $g(n)$.
- (c allows us to shift $g(n)$ up by a constant amount to account for machine speed, etc.)
- [Introduced in 1894 by Paul Bachmann, popularized by Don Knuth.]

O-notation

- Example: To show that $\frac{1}{2}n^2 - 3n = O(n^2)$

- If $f(n)$= Θ(g(n)), then $f(n)$= O(g(n)).
- Typically, in this class, we'll want things to be *sub-linear*: we don't want to look at every data item.
  - Quadratic function: $O(n^2)$
  - Linear function: O(n)
  - Sublinear function: O(log n), $O(n^{1/2})$, $O(n^{0.99})$,…

Alternative Definition of O-notation

- O( ) notation focuses on the largest term and ignores constants
  - Largest term will dominate eventually for large enough *n*.
  - Constants depend on "irrelevant" things like machine speed, architecture, etc.
- Definition: $f(n)$ is $O(g(n))$ if the limit of $f(n) / g(n)$, is a **constant** as n goes to infinity.

Examples

- Example 1:
  - Suppose f(n) = $12n^2 + n + 2 \log n$.
  - Consider $g(n) = n^2$
  - Then lim $[f(n)/g(n)]$= lim$[12 + (1/n) + (2 \log n) / n^2] = 12$

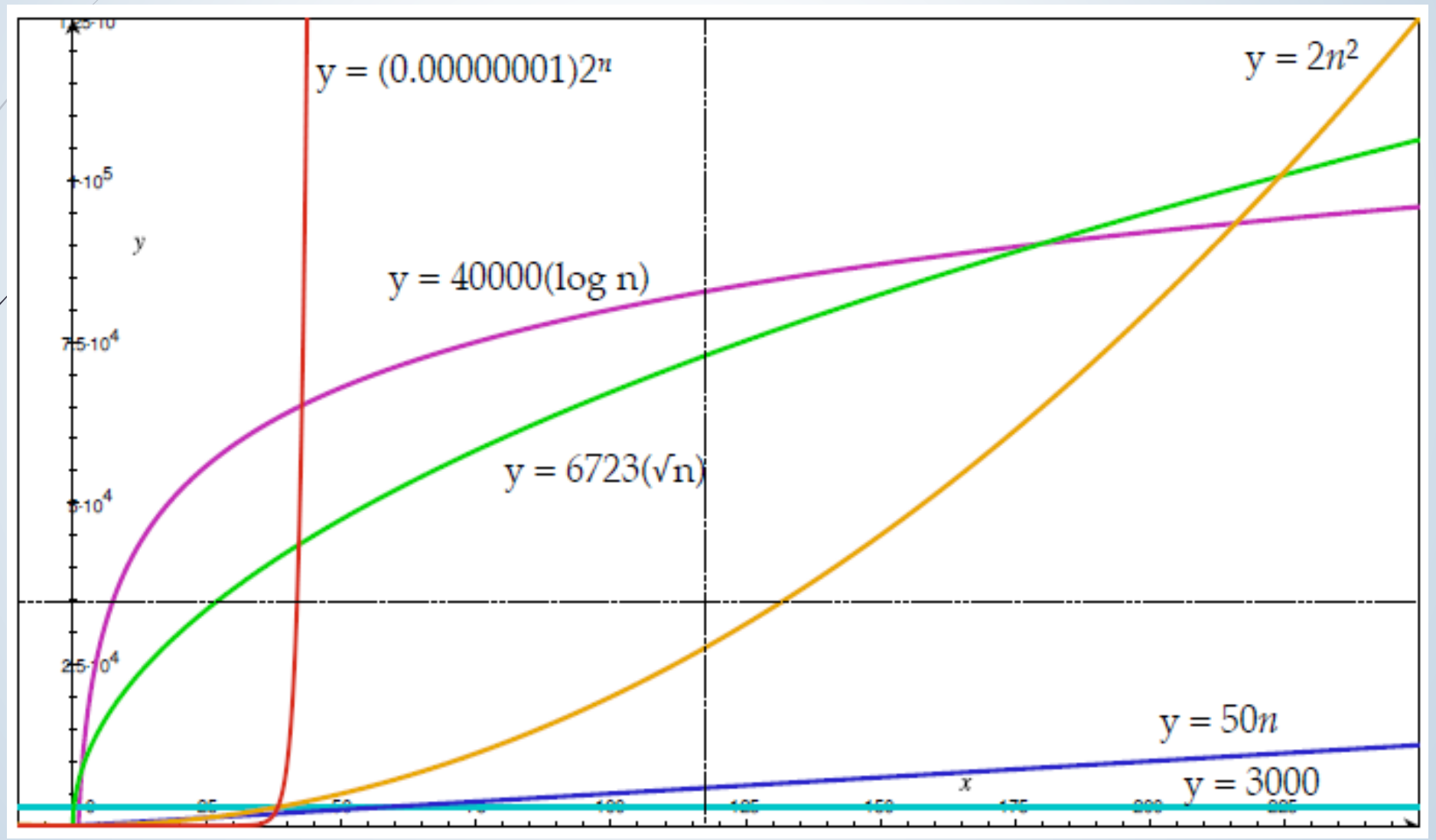
$f(n)$= O(g(n))
or f(n)=O(n^2)

- Example 2:
  - Is f(n) = n log n in O(n) ?
  - Check: lim $[(n \log n) / n]$ = lim $\log n$ = infinity! So no!

Big-O Taxonomy

Growth Rate	Name	Notes
$O(1)$	constant	Best, independent of input size
$O(\log \log n)$		very fast
$O(\log n)$	logarithmic	often for tree-based data structures
$O(\log^k n)$	polylogarithmic	
$O(n^p)$, $0 < p < 1$	E.g. $O(n^{1/2}) = O(\sqrt{n})$	Still sub-linear
$O(n)$	linear	Have to look at all data
$O(n \log n)$		Time to sort
$O(n^2)$	quadratic	Ok if n is small enough
$O(n^k)$	polynomial	Tractable
$O(2^n)$	exponential	bad
$O(n!)$	factorial	bad

Big-O Examples

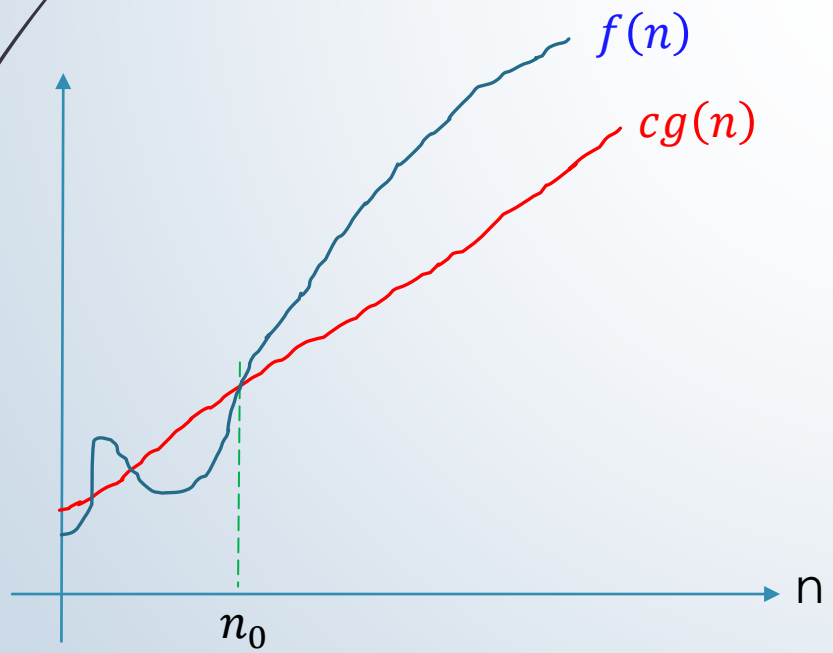


Ω-notation

- Ω-notation provides an asymptotic lower bound.

For a given function g(n), we denote by Ω(g(n)) the set of functions

Ω(g(n))={f(n): there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n $\geq n_0$}



$f(n)$= Ω(g(n))

Ω-notation

- Example: To show that $\frac{1}{2}n^2 - 3n = \Omega(n^2)$

- If $f(n)$= Θ(g(n)), then $f(n)$= Ω(g(n)).
- If $f(n)$= Ω(g(n)) and $f(n)$= O(g(n)), then $f(n)$= Θ(g(n)).

Asymptotically tight or not?

- $2n=O(n)$ this is asymptotically tight
- $2n=O(n^2)$ this is **not** asymptotically tight → o-notation

$o(g(n))=\{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$n=\Omega(\log n)$ this is not asymptotically tight
$n=\Omega(n)$ this is asymptotically tight → ω-notation

$\omega(g(n))=\{f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

100n versus n^2 100n versus n

- Why 100n=O(n) ?
  - When n is very big like 10000000000000000
  - 100n is 1000000000000000000
  - n is 10000000000000000
  - nearly the same
- Why 100n=o(n^2) ?
  - When n is 10000000000000000
  - 100n is 1000000000000000000
  - n^2 is 100000000000000000000000000000000
  - Differ a lot!

Example

$n \log n = o(n^2)$

$n \log n = O(n \log n)$ but $n \log n \neq o(n \log n)$

$2n = o(n^2)$

$2n = O(n)$ but $2n \neq o(n)$

$2n^2 = o(n^3)$

$2n^2 = O(n^2)$ but $2n^2 \neq o(n^2)$

Asymptotic Notation

- Relationship between typical functions
  - log n = o (n)
  - n = o (n log n)
  - n^c = o (2^n) where n^c may be n^2, n^4, etc.
  - If f(n)=n+log n, we call log n lower order terms

  (You are not required to analyze, but remember these relations)

  log n < $\sqrt{n}$ < n < nlog n < n^2 < n^4 < 2^n < n!

Asymptotic Notation

- When calculating asymptotic running time
  - Drop low-order terms
  - Ignore leading constants
- Example 1: $T(n) = An^2+Bn+C$
  - An^2
  - $T(n) = O(n^2)$
- Example 2: $T(n) = Anlogn+Bn^2+Cn+D$
  - Bn^2
  - $T(n) = O(n^2)$

Exercises

1. Write the order of the following functions with O-notation (asymptotically tight bound).
   - $f(n)=2n^2+3n+5$
   - $g(n)=1000n\log n+5$

2. True or False
   - Is $2^{n+1} = O(2^n)$?
   - Is $2^{2n} = O(2^n)$?

3. Why "The running time of algorithm A is at least $O(n^2)$" is meaningless ?

Asymptotic Performance

Very often the algorithm complexity can be observed directly from simple algorithms

```
Insertion-Sort(A)
1   for j = 1 to n-1
2       key = A[j]
3       i = j-1
4       while i >= 0 and A[i] > key
5           A[i+1] = A[i]
6           i = i - 1
7       A[i+1] = key
```

$O(n^2)$

There are 4 very useful rules for such Big-Oh analysis ...

Asymptotic Performance

General rules for Big-Oh Analysis:

Rule 1. FOR LOOPS

The running time of a *for* loop is at most the running time of the statements inside the *for* loop (including tests) times no. of iterations

```
for (i=0;i<N;i++)
        a++;
```

O(N)

Rule 2. NESTED FOR LOOPS

The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for (i=0;i<N;i++)
        for (j=0;j<N;j++)
                k++;
```

$O(N^2)$

Rule 3. CONSECUTIVE STATEMENTS

Count the maximum one.

```
for (i=0;i<N;i++)
        a++;

for (i=0;i<N;i++)
        for (j=0;j<N;j++)
                k++;
```

$O(N^2)$

Rule 4. IF / ELSE

For the fragment:

```
If (condition)
        S1
else
        S2,
```

take the test + the maximum for S1 and S2.

Asymptotic Performance

Example of Big-Oh Analysis:

```
void function1(int n)
{  int i, j;
   int x=0;

   for (i=0;i<n;i++)
         x++;

   for (i=0;i<n;i++)
      for (j=0;j<n;j++)
         x++;
}
```

This function is O(__)

```
void function2(int n)
{  int i;
   int x=0;

   for (i=0;i<n/2;i++)
         x++;
}
```

This function is O(__)

Asymptotic Performance

Example of Big-Oh Analysis:

```
void function3(int n)
{   int i;
    int x=0;

    if (n>10)
        for (i=0;i<n/2;i++)
            x++;
    else
    {   for (i=0;i<n;i++)
            for (j=0;j<n/2;j++)
                x--;
    }
}
```

This function is O(__)

```
void function4(int n)
{   int i;
    int x=0;

    for (i=0;i<10;i++)
        for (j=0;j<n/2;j++)
            x--;
}
```

This function is O(__)

Asymptotic Performance

Example of Big-Oh Analysis:

```
void function5(int n)
{  int i;
   for (i=0;i<n;i++)
      if (IsSignificantData(i))
         SpecialTreatment(i);
}
```

Suppose
IsSignificantData is O(n),
SpecialTreatment is O(n log n)

This function is O(____)

What is the complexity of the following algorithm ?

```
Insertion-Sort(A)
1   for j = 1 to n-1
2       key = A[j]
3       i = j-1
4       while i >= 0 and A[i] > key
5           A[i+1] = A[i]
6           i = i - 1
7       A[i+1] = key
```

Let's check the three algorithms for Top N!

Algorithm 1: Search n times

1. Save all the scores in an array A[m]
2. for k=1 to n
3. Find the largest number in A and output it
4. Remove the number found in step 3 from A.

O(m*n)

Algorithm 2: Sorting first

1. Save all the scores in an array A[m]
2. Sort A in decreasing (non-increasing) order.
3. Output the first n elements of A in order.

if step 2 sorting algorithm's complexity is g(n)

O(m+g(n)+n)

Algorithm 3: Counting frequency

1. Count the frequency of number p and save it in array C[p]
2. Check array C in the order of C[M], C[M-1], C[M-2],….

   If C[p]>0, output p for C[p] time(s) until we totally output n numbers.

- where M is the largest number.

O(m+M+n)

Maximum Profit

Time Limit : 1 sec, Memory Limit : 131072 KB

Maximum Profit

You can obtain profits from foreign exchange margin transactions. For example, if you buy 1000 dollar at a rate of 100 yen per dollar and sell them at a rate of 108 yen per dollar, you can obtain (108 - 100) × 1000 = 8000 yen.

Write a program which reads values of a currency R_t at a certain time t (t = 0, 1, 2, ... n-1), and reports the maximum value of $R_j - R_i$ where j > i .

Input

The first line contains an integer n. In the following n lines, R_t (t = 0, 1, 2, ... n-1) are given in order.

Output

Print the maximum value in a line.

Constraints

2 <= n <= 200,000

1<= R_t <= 1,000,000,000

Sample Input 1

```
6
5
3
1
3
4
3
```

Sample Output 1

```
3
```

Sample Input 2

```
3
4
3
2
```

Sample Output 2

```
-1
```

Source: https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_1_D

Method 1: Simple algorithm

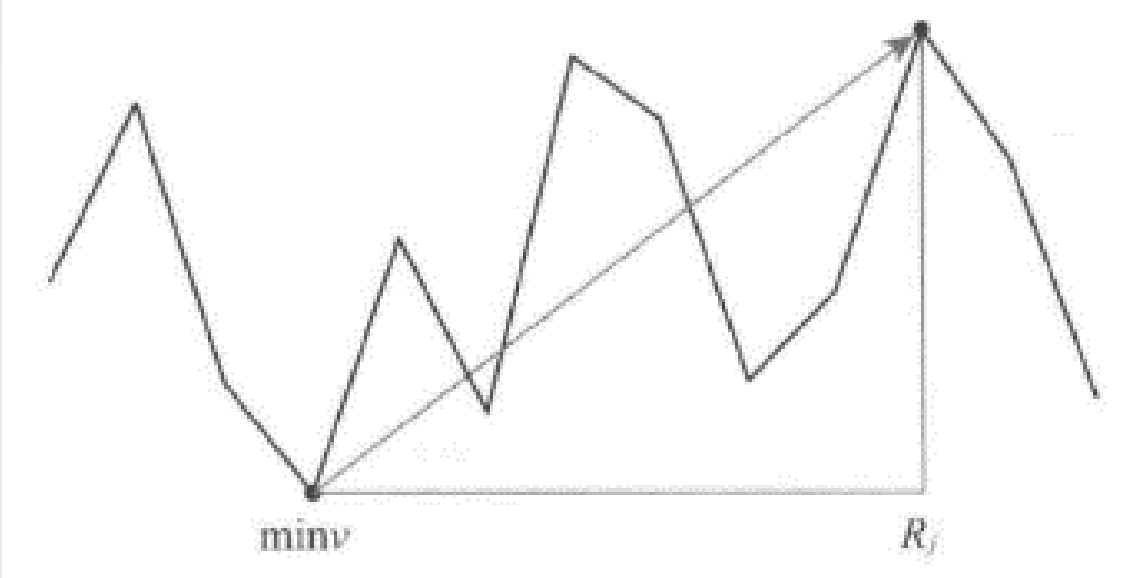
- Let maxv be the maximum profit. We can use the following simple algorithm to get the solution.

```
// Simple algorithm
for j from 1 to n-1
      for i from 0 to j-1
            maxv=max{maxv,  R[j]-R[i]}
```

Note: the initial value of maxv should be small enough, or you can use R1-R0 as the initial value.

- What is the complexity of this simple algorithm ?

Method 2: Quick algorithm




```
//Quick algorithm
minv=R[0]
for j from 1 to n-1
        maxv=max{ maxv,  R[j]-minv }
        minv=min{ minv, R[j]}
```

What is the complexity of this quick algorithm ?

Recursion

- A function or procedure that calls itself.
- Need to beware of infinite loop

```
#include<bits/stdc++.h>
using namespace std;
//Get the sum of the first n integers.


int sum(int n){
    if (n==1)
        return 1;
    else
        return n+sum(n-1);
}


int main(){
    int n;
    cin>>n;
    cout<<sum(n);
}
```

Base case

recurring case

Recursion

- Can simply the code
- Can solve the problems with the following properties:
  - The problem can be divided or reduced into the same problem with small parameters
  - We need information of the subproblems to solve the current one.

Example: factorial

n's factorial: n!=1*2*3*…*n

```
int factorial(int n){
    if (n==1)
        return 1;
    else
        return n*factorial(n-1);
}
```

Make sure that you have an end.

Complexity

```
factorial(n)
    if n==1
        return 1
    else
        return n*factorial(n-1)
```

T(n)=T(n-1)+c
T(1)=O(1)

T(n)=T(n-1)+c
=(T(n-2)+c)+c
=(T(n-3)+c)+c+c
…
=T(n-k)+k*c
…
=T(1)+(n-1)*c
=c'+(n-1)*c
=O(n)

Example: GCD

- The greatest common divisor (GCD) of two or more numbers is **the greatest common factor number that divides them, exactly**. It is also called the highest common factor (HCF).
- Most common way of computing GCD quickly is by Euclidean algorithm.

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Example: Fibonacci Number

Base case: F(0)=1, F(1)=1

Recursive relation: F(n)=F(n-1)+F(n-2) if n>1

```
int f(int n){
    if (n<2)
        return 1;
    else
        return f(n-1)+f(n-2);
}
```

Warning: **DO NOT** use recursion to calculate the n-th Fibonacci number using recursion **without** memorization as it is very slow

Example: Hanoi Tower

Move n plates from pile A to pile C via pile B. Small plates can never be placed under larger plates.

```
Hanoi(n, A, B, C)
    If n==1
            move the plate from A to C directly
    else
            move n-1 plates from A to B via C
            move 1 plate from A to C
            move n-1 plates from B to C via A
```

Divide and Conquer

- **Divide** the problem into smaller local problems
- Recursively **solve** the local problem
- **Conquer** the global problem by combining the results of local problems.

Find the minimum value

```
findMaximum(A, left, right)
   mid=(left + right) / 2          // Divide
   if left== right                 //  There is only one element
        return A[left]
   else
        u=findMaximum(A, left, mid)  //solve the first half
        v=findMaximum(A, mid+1,right)   //solve the second half
        x=max(u,v)                    // Conquer
        return x
```

Complexity

```
findMaximum(A, left, right)
   mid=(left + right)/2
   if left== right
        return A[left]
   else
        u=findMaximum(A, left, mid)
        v=findMaximum(A, mid+1,right)
        x=max(u,v)
        return x
```

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

$T(1)=O(1)$

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

$=2(2T(n/4)+c)+c$

$=4(2T(n/8)+c)+c+2c$

…

$=2^kT(n/2^k)+c*(1+2+2^2+\ldots+2^{k-1})$

$=2^kT(n/2^k)+c*(2^k-1)$

…

$=2^{\log n}T(n/2^{\log n})+(2^{\log n}-1)*c$

$=nT(1)+(n-1)*c$

$=n*c'+(n-1)*c$

$=O(n)$

Example: Power

- Compute $a^b \bmod P$ where a and b are both large integers and P=1000000007 which is a prime.

```
#define P 1000000007
long long power(long long a, long long b){
    if(b==0){
        return 1;
    }
    else{
        if(b%2==0){
            return power((a*a)%P, b/2);
        }
        else{
            return (power((a*a)%P, (b-1)/2)*a)%P;
        }
    }
}
```