# Thomas Jefferson Invitational Open in Informatics

**Study Guide**

**Version 2.012.05**

***By programmers, For programmers***

# Preface

Greetings!

I congratulate you on all your achievements thus far – computer science is a field like no other. As I'm sure you have already noticed, computer science is neither simple nor easy. However, despite the challenges, your intellectual curiosity has brought you this far and will continue to lead you through a very successful life. As you progress through your computer science curriculum, you will begin to notice subtleties in the various topics you will explore. Some of you will be intrigued by computer graphics; others will be interested in artificial intelligence; still others of you will gravitate towards network design and maintenance, and these are only a few topics that may grab your attention in the vast universe of computer science.

Currently, you are still a student beginning to learn about the fundamental concepts and theories in computer science; in fact, I am also still a student learning the same knowledge as you. But, something drew you and me toward the field and subfields of computer science: our natural curiosities. These curiosities have motivated both you and me to dig deeper and chip away at the complexities in computer science. Sometimes, we can satisfy these curiosities using what we have learned in class: recalling and applying the various tactics and techniques we have read about and completed assignments on. But, what about the questions that cannot be easily answered by what you currently have?

That is probably a reason why you picked up this study guide. *A* reason, not *the* reason. What you will find in the following pages is a series of relatively comprehensive materials designed to supplement what you have learned in class. You might already be familiar with several of the topics presented, or you might not. However much you know about these topics, be prepared to apply them in clever ways to catch and solve the various problems that will be presented to you at the competition, and possibly in your future career.

Finally, I will address a point that is of great importance in computer science: the value of teamwork. No matter how brilliant *you* may be, you can only think about so much by yourself. In reality, very little computer science is done by a lone programmer mashing the keyboard in a dark room piled high with boxes of pizza. Similarly, we want you to work with a team, as a team, to tackle the problems presented to you. In fact, the entire competition is a team competition, so that you can work together as a team from the moment you start preparing, through the day of the contest, and onwards as you learn about each other's strengths and abilities.

With that, I wish for you to have an enjoyable experience in the next few months, and I hope to see you at the competition!

Sincerely,

Luo Qian
TJ IOI Chief Executive Officer

# The TJ IOI Contest

The contest itself is split into two portions: the Theoretical (free response) and Practical (programming) portions. You will work with your team for both of these portions, and as such, will be scored as a team.

The Theoretical part of the contest is a 60-minute written exam of 15 multi-part questions. Questions range from relatively easy difficulties to very challenging problems. During this part of the contest, you will be expected to work with the rest of your team to complete the entire examination. Solutions should be written cleanly, neatly, and legibly in order to receive credit. Moreover, teams with particularly clever (and correct) solutions are likely to receive slightly more credit than a less elegant solution; however, incorrect solutions will be marked down regardless of cleverness.

The Practical portion of the contest is a 150-minute programming competition. You will be given 10 programming problems, ranging in difficulty from easy to extremely challenging. Your team will be provided with 2 Linux-system computers (see Linux Guide), and the programming languages Java, C/C++, and Python. Submissions will be subjected to a series of test cases and will interact with the grading system through the use of file input and output. Correct outputs that run within the given time limit and do not exceed the memory limits will be awarded points, while incorrect outputs will not be awarded points. Additionally, programs with hardcoded solutions (e.g. an inordinate amount of `if` statements) will be penalized at the judges' discretions.

Both portions of the contest will require you to apply the knowledge you have gained in your studies and through these study materials. Each team's overall score is a composite of its scores on the Theoretical and Practical portions of the contest. Therefore, the overall winners of the competitions must demonstrate impressive knowledge of computer science both on a theoretical level and on a practical level. Additionally, teams that are well-organized and cohesive will find the competition to be less difficult than teams that are not well-organized or cohesive. Therefore, it is in your best interest to work out a well-established team plan before the actual competition.

This is a thrilling contest, and we hope that you are as excited as we are!

## Contest Schedule

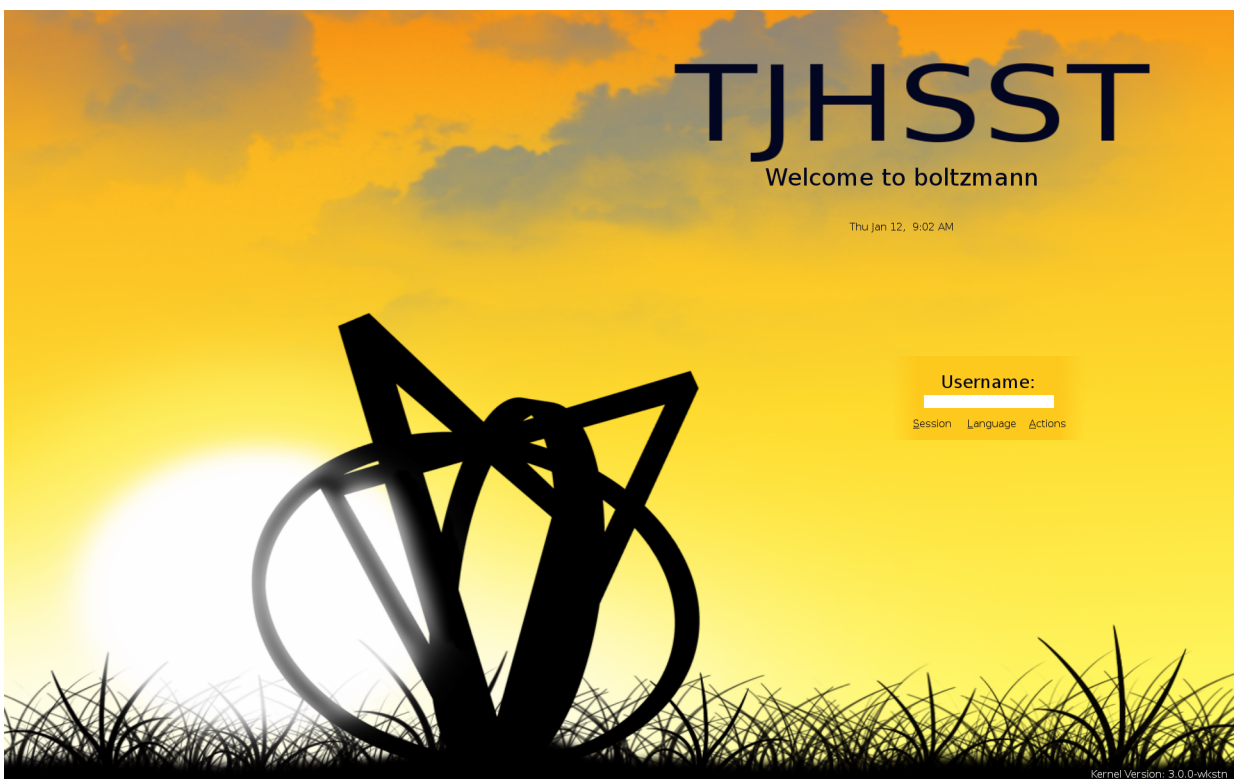
Below is the planned schedule of events for the contest. This is also available on the TJ IOI website (`http://activities.tjhsst.edu/tjioi/?schedule`).

| Time | Activity |
|---|---|
| 9:30AM | Teams arrive and sign in |
| 10:00AM–10:15AM | Opening ceremony and remarks |
| 10:20PM–11:20AM | Part I (Theoretical – Short Answer) |
| 11:20AM–12:20PM | Lunch |
| 12:20PM–1:20PM | Systems Orientation and Practice |
| 1:35PM–4:05PM | Part II (Practical – Programming) |
| 4:10PM–4:25PM | Break and transition |
| 4:30PM–5:00PM | Awards ceremony |

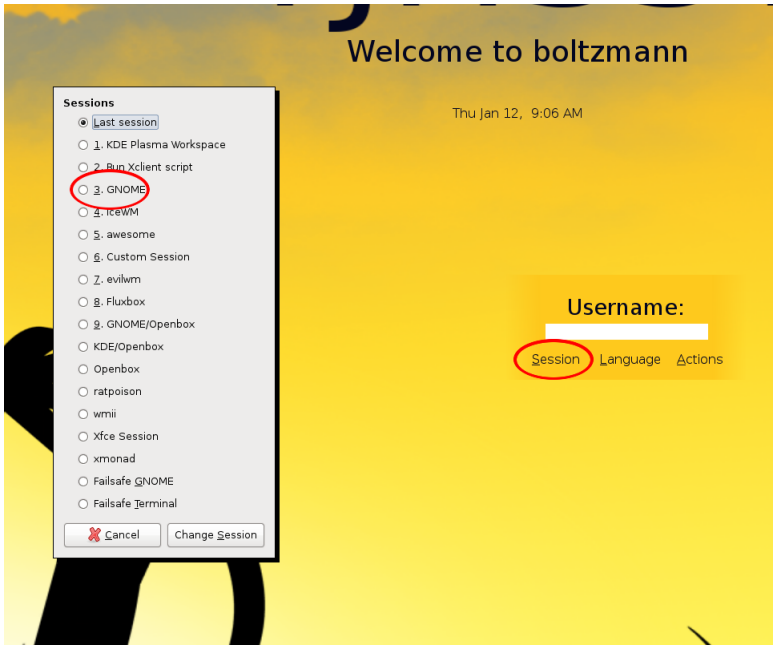
## Linux Guide

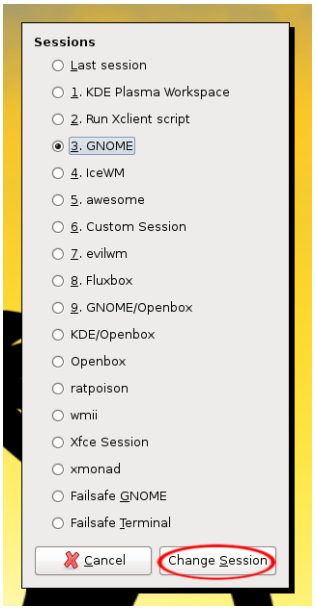
This is a quick guide to introduce you to the basic Linux functionalities that you might encounter at the competition.

Below is the login screen for the machines at the TJ Computer Systems Lab (CSL).

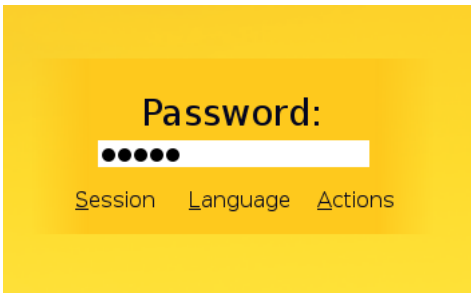


Before logging on for the first time, you may wish to switch your desktop manager to something closer to a Windows one. As such, we recommend the GNOME desktop manager, which is exceedingly user-friendly.

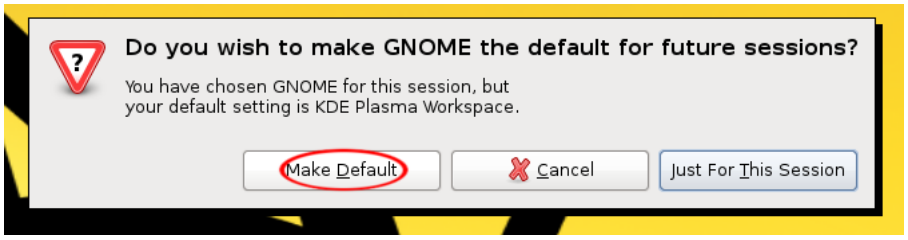


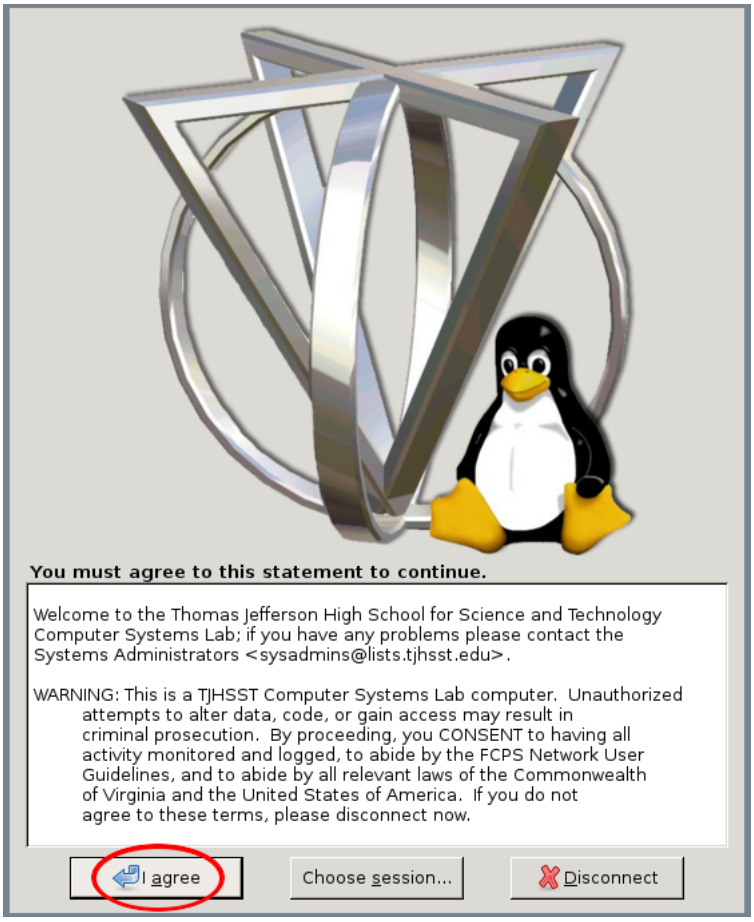

Then, enter your username (replace the "##" with your team's 2-digit ID, e.g. 00 or 12), press the Enter key, enter your password (which will be given to you), and press the Enter key again. See screenshots below.




You may then be prompted to confirm whether or not to make your selected session the default session. You probably want to choose "Make Default" here.



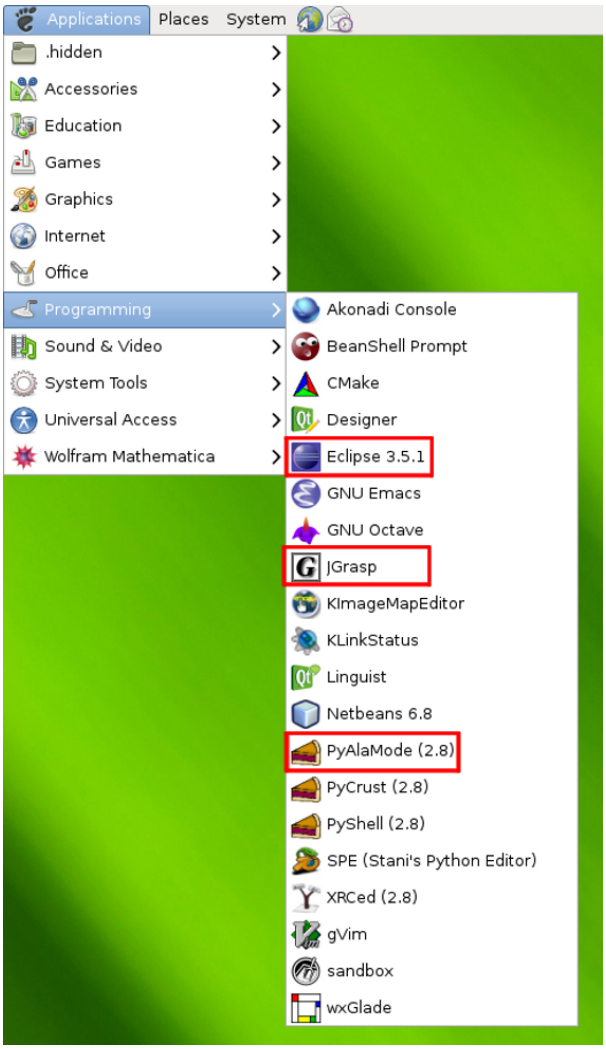
Now, another dialog should pop up (and will for each time that you log in). If you want to know the details, feel free to read the contents in their entirety. However, if you want to continue, click the "I Agree" button.

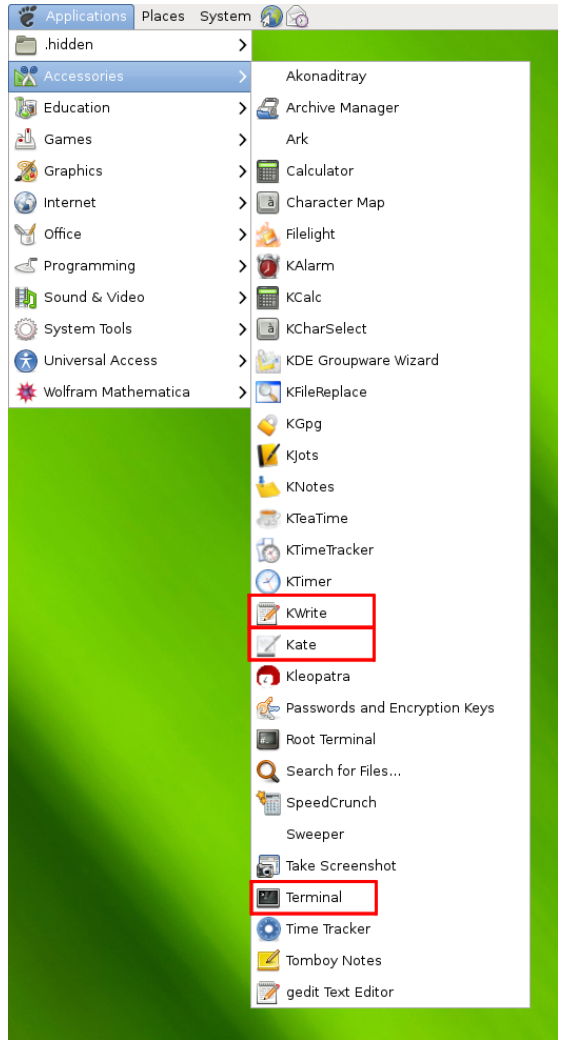



Once you agree to the terms and conditions of using the CSL, you will be logged in and your desktop manager should appear. You might be prompted to change your password.

## Applications

There are quite a few programs for you to use: Eclipse, JGrasp, and PyAlaMode are all under Applications→Programming, while KWrite, Kate (KDE Advanced Text Editor) and Terminal (the Linux command line) are available under Applications→Accessories.



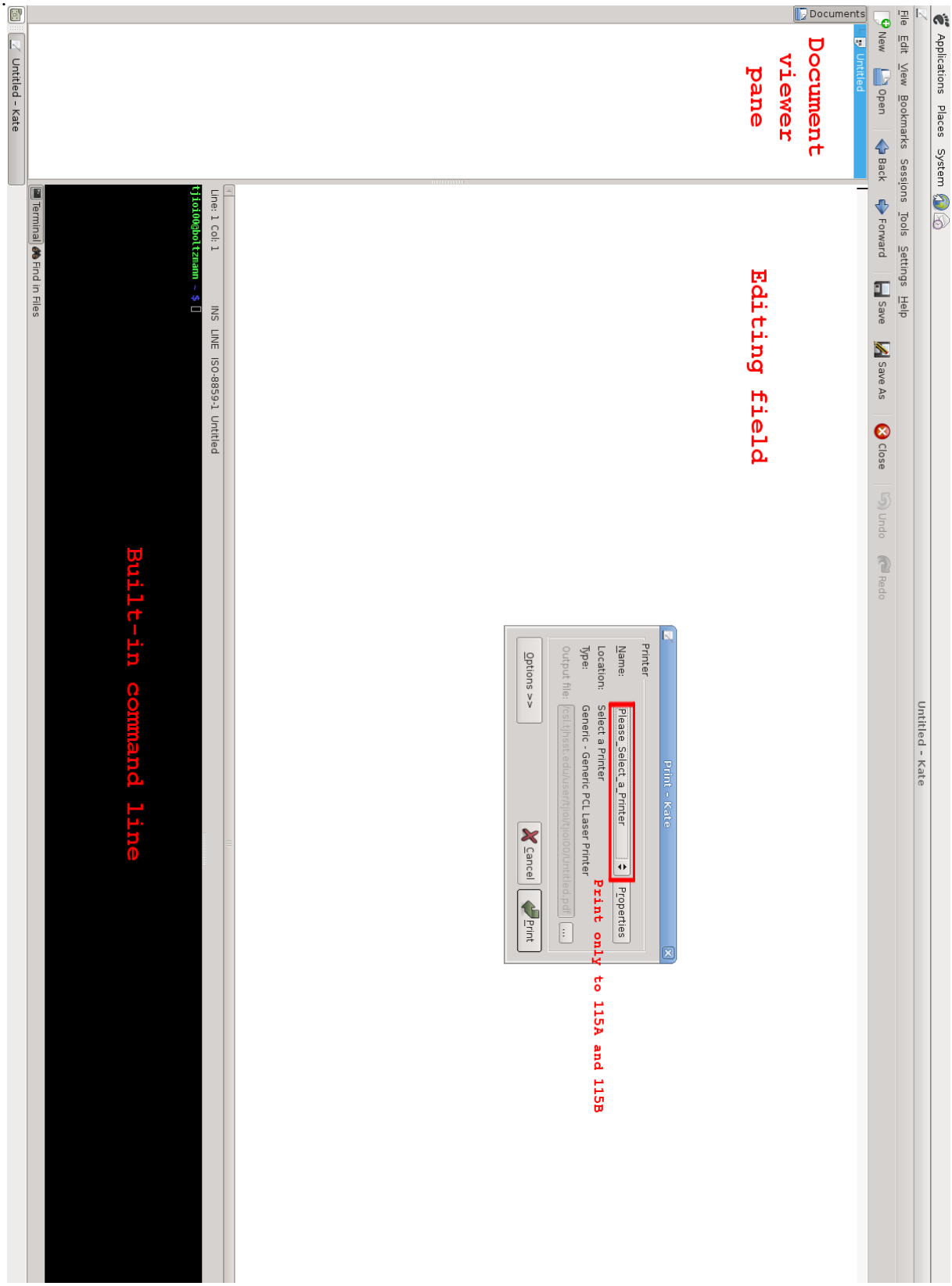




Below, we have included a noncomprehensive guide to using Kate and the Linux command line.

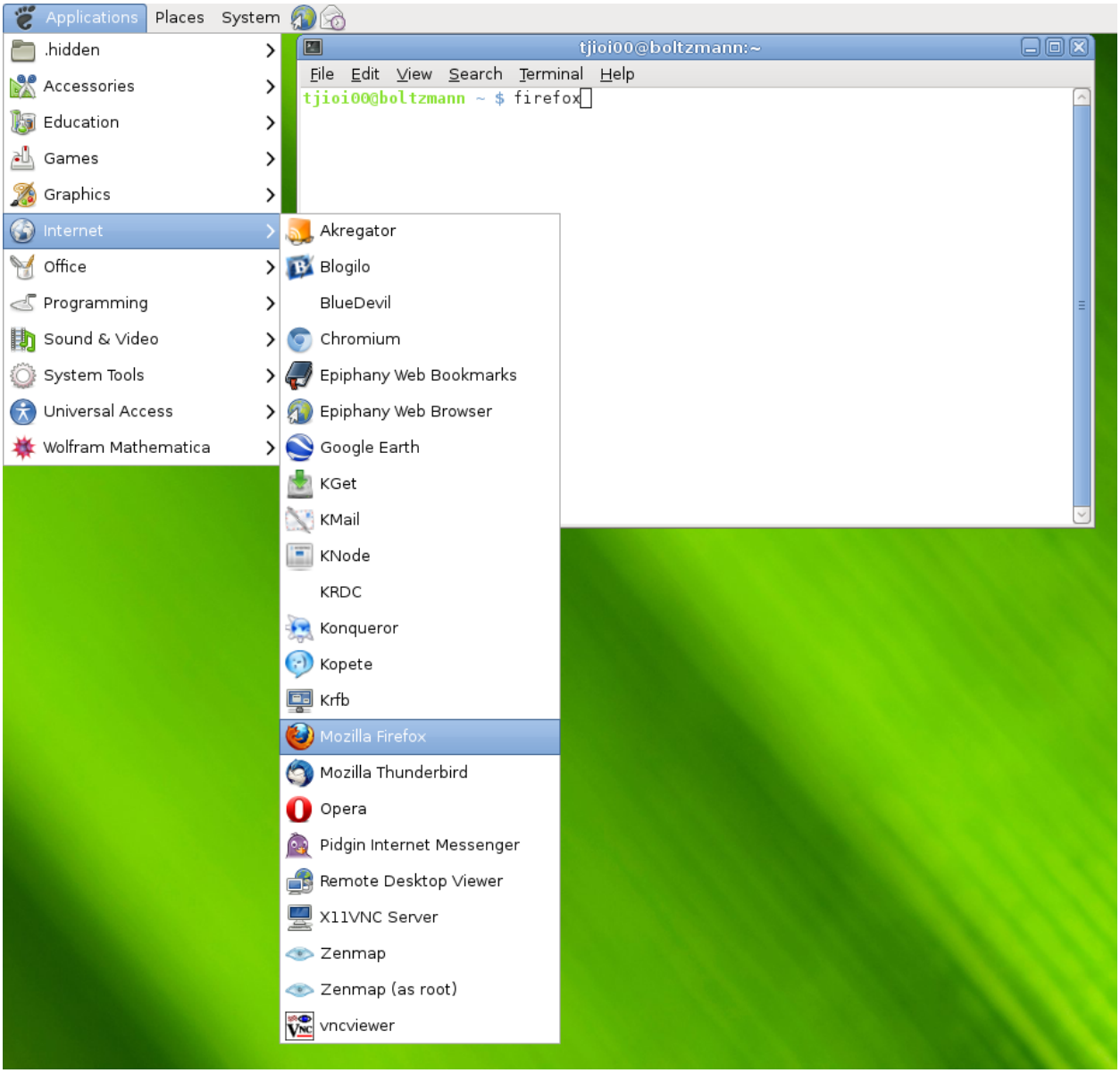
The Linux command line:

```
Applications  Places  System                                                    tjioi00@boltzmann:~
File  Edit  View  Search  Terminal  Help
tjioi00@boltzmann ~ $ ls   List the contents of the current directory
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos   The list of the contents
tjioi00@boltzmann ~ $ cd Desktop/   cd (directory) changes the current directory to (directory)
tjioi00@boltzmann ~/Desktop $ ls
tjioi00@boltzmann ~/Desktop $ mkdir ioi   mkdir (dir) makes a directory called (dir) in the current directory
tjioi00@boltzmann ~/Desktop $ ls
ioi
tjioi00@boltzmann ~/Desktop $ cd ioi
tjioi00@boltzmann ~/Desktop/ioi $ ls
tjioi00@boltzmann ~/Desktop/ioi $ touch asdf.cpp   (not needed) touch (name) creates a file called (name)
tjioi00@boltzmann ~/Desktop/ioi $ ls
asdf.cpp
tjioi00@boltzmann ~/Desktop/ioi $ cd ../   ../ means the parent directory, while ./ means the current directory
tjioi00@boltzmann ~/Desktop $ rmdir ioi   rmdir (dir) removes the directory dir; only works if dir is an empty directory
rmdir: failed to remove `ioi': Directory not empty
tjioi00@boltzmann ~/Desktop $ cd ioi
tjioi00@boltzmann ~/Desktop/ioi $ rm asdf.cpp   rm (file) removes a file; does not work on directories
tjioi00@boltzmann ~/Desktop/ioi $ ls
tjioi00@boltzmann ~/Desktop/ioi $ cd ..
tjioi00@boltzmann ~/Desktop $ rmdir ioi
tjioi00@boltzmann ~/Desktop $ ls
tjioi00@boltzmann ~/Desktop $ cd ~   ~ is the macro for your home directory
tjioi00@boltzmann ~ $
```

Kate:


Applications Places System
Untitled - Kate
File Edit View Bookmarks Sessions Tools Settings Help
New Open Back Forward Save Save As Close Undo Redo
Untitled
Documents
Untitled - Kate
Document viewer pane
Editing field
Print - Kate
Printer
Name: Please_Select_a_Printer
Properties
Location: Select a Printer
Type: Generic - Generic PCL Laser Printer
Output file: /cal.hjhsst.edu/user/tjio/tjio00/Untitled.pdf
Options >>
Cancel
Print
Print only to 115A and 115B
Line: 1 Col: 1 INS LINE ISO-8859-1 Untitled
tjio00@boltzmann ~ $
Terminal
Find in Files
Built-in command line

Finally, there are two ways to launch web browsers. The first is to select it from Applications→Internet, and the second is to type the name of the browser into the command line.




## Conclusion

With that, we will conclude this Linux guide. Be aware that we will have plenty of time to familiarize everyone with the system (and with the online submissions portal) on the morning of the contest, as this is part of the orientation process. Now, without further ado, the table of contents is on the next page.

# Contents

**Preface** **i**

**The TJ IOI Contest** **ii**
Contest Schedule . . . iii
Linux Guide . . . iii

**Mathematics in Computer Science** **1**
Computer Number Formats . . . 1
Data Types . . . 2
Recursion . . . 3
Order of Operations . . . 5

**Object-Oriented Programming** **7**

**Code Theory** **9**
Code Analysis . . . 9
Code Writing . . . 10
Code Debugging . . . 11

**Boolean Algebra** **13**
Operators . . . 13
Truth Tables . . . 13
Identities . . . 14

**Code Efficiency†** **15**
Big O Notation . . . 15
Determining Big O . . . 16
Common Big O Values . . . 16

**Data Structures†** **17**
Linear Data Structures (Lists, Stacks, and Queues) . . . 17
Tree-Based Data Structures . . . 18

**Graph Theory†** **21**

**Search Techniques†** **24**
Binary Search . . . 24
Depth First Search . . . 24
Breadth First Search . . . 25
DFS or BFS? . . . 26

**Dynamic Programming†** **27**
What is Dynamic Programming? . . . . . 27
Knapsack Algorithm . . . . . 27
General Strategy . . . . . 29

**Changelogs** **31**

*Note: Section titles are clickable links to their respective sections.

†Advanced sections: These will account for $\frac{1}{4}$ of the total contest. They should only be attempted if the other sections have been mastered.

# Mathematics in Computer Science

Math and computer science go hand-in-hand. Here, we will focus on the more mathematical aspects of computer science, including number formats, data types, recursion, and the CS order of operations. Because these topics are crucial to any program (and language), every computer scientist should be familiar with them.

## Computer Number Formats

Some problems on the contest might relate to numbers in other bases, which are common in the computing world. A number's **base** or **radix** indicates how many digits are in the number system. When we count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, etc., we are counting in base 10. However, it is possible to count in other bases too, such as in binary (base 2): 0, 1, 10, 11, 100, 101, etc. This might seem strange at first, but it makes mathematical sense.

In base 10, we have exactly 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. When we have the number 1234, it means $1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$. Similarly, base 2 has only two digits: 0 and 1. The number 1010 in base 2 (written as $1010_2$) can be converted to base 10 by calculating: $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 2 = 10$. Thus, $1010_2 = 10_{10}$.

We can have bases greater than 10 as well. Commonly seen is hexadecimal, which is base 16. There are 16 "digits": 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. When counting, we count normally but go from 9 to A, then to B, C, D, E, and F, and then finally 10.

One common trick for converting between bases $b$ and $c$ where $c = b^n$, for $n$ is an integer, is as follows:

1. First, we compare a number $k$'s representation in bases $b$ and $c$: $k_b$ and $k_c$, respectively.
2. Because $c = b^n$, the remainder of $\frac{k_b}{k_c}$ is the last $n$ digits of $k_b$, which corresponds to the last digit of $k_c$.
3. Note that this process can be continued recursively, as it is infinitely applicable to each block of $n$ digits in $k_b$.
4. Thus, converting $k_b$ from base $b$ to base $c$ can be done in chunks of $n$ digits at a time.
5. This process can be reversed if converting from $c$ to $b$.

For an example, please refer to sample problem #3 below.

Sample Problem #1:

Convert $212_3$ to decimal.

This is $2 \cdot 3^2 + 1 \cdot 3^1 + 2 \cdot 3^0 = 2 \cdot 9 + 1 \cdot 3 + 2 \cdot 1 = 18 + 3 + 2 = 23$.

Answer: **23**

Sample Problem #2:

Convert $200_{10}$ to base 3.

We want to find a number $abcde_3$, where each letter represents a single digit in base 3, such that $abcde_3 = 200_{10}$. That is, $a \cdot 3^4 + b \cdot 3^3 + c \cdot 3^2 + d \cdot 3^1 + e \cdot 3^0 = 200$ (in base 10). We know that there are at most 5 digits because $3^5 = 243$, which does not fit into 200.

The highest power of 3 here, $3^4 = 81$ goes into 200 a total of 2 times. Thus, we can start our base 3 number like this:

$200_{10} = 2????_3$

After putting 81 two times into 200, we have $200 - 162 = 38$ leftover. The next power of 3 is $3^3 = 27$, which goes into 38 once. 1 is the next digit. We have 11 left over. This can be represented as $9 + 2 = 3^2 + 2$ (note that $2 = 2 \cdot 3^0$). Our final answer is:

Answer: $\mathbf{21102_3}$

Sample Problem #3:

Convert $FEED_{16}$ to base 2.

$FEED_{16}$ in base 10 equals $F \cdot 16^3 + E \cdot 16^2 + E \cdot 16^1 + D \cdot 16^0 = 15 \cdot 4096 + 14 \cdot 256 + 14 \cdot 16 + 13 = 65261$. This is really annoying to convert back to base 2. Instead we can note that $16 = 2^4$. This means $FEED_{16} = F \cdot 2^{12} + E \cdot 2^8 + E \cdot 2^4 + D = 1111_2 \cdot 2^{12} + 1110_2 \cdot 2^8 + 1110_2 \cdot 2^4 + 1101_2 = 1111\ 1110\ 1110\ 1101_2$.

Answer: $\mathbf{1111111011101101_2}$

## Data Types

Most programming languages are typed, and as such, require the use of type variables. However, because there are limitations on physical memory, these variables are allocated only so much memory. This leads to limits on the maximums and minimums for storage values.

Before we begin that, we must quickly review some memory terminology. First, a **bit** is a binary digit, the most basic unit of memory: a 1 or 0. Thus, one solo bit can only represent a 1 or 0. If we take 8 bits together, we form a **byte**, which is another unit of measurement for memory. Because a byte can store 8 binary digits, it can represent $2^8$ distinct values: $00000000_2$ to $11111111_2$. Also, when indexing bits, the first bit on the right is indexed at 0, and progresses to the left. Thus, the $n^{th}$ index bit represents the coefficient for $2^n$.

The most common data type is probably the `int`. `int` stands for "integer," and is generally (almost always, in fact) a 4-byte integer value. However, the range of numbers that an int can hold does not run from 0 to $2^{32}$-1; instead, the last bit in an integer is used to determine whether or not the value is negative. If the last bit (index 31) is 1, the number is negative; if it is 0, the number is nonnegative. This presence of the signed bit makes the integer a **signed** `int` (there are unsigned `int`s, but they will not be on this contest). Also, "negative 0" actually represents $-2^{31}$. This means that the range of numbers (letting "negative 0" equal $-2^{31}$) for a signed `int` is from $-2^{31}$ to $2^{31} - 1$ (inclusive). This phenomenon, of allowing either the positive or negative spectrum to be longer than the other, is called **bias**.

A table of the sizes and ranges of some common data types is below. Note that Python, being a loosely typed language, does not have explicit data types, and as such, does not appear in the table:

| **Name** | **Size(bytes)** | **Range** | **Java** | **C++** |
|---|---|---|---|---|
| boolean | 1 | false (0), true (not 0) | `boolean` | `bool` |
| character | 1 | $[-128, 127]$ ($[-2^7, 2^7 - 1]$) | `char` | `char` |
| integer | 4 | $[-2^{31}, 2^{31} - 1]$ | `int` | `int` |
| long [integer] | 8 | $[-2^{63}, 2^{63} - 1]$ | `long` | `long long` |

Floating-point numbers are similarly represented. However, unlike integer representations, a floating-point data type is split into three segments: the sign bit (last bit), exponent bits, and significand (first few bits). The exponent bits represent the exponent of 2; the significand represents the number itself. Additionally, the significand is processed as a number $n$ such that $1 \leq n < 10$. In this way, floating-point numbers are somewhat similar to scientific notation. Also, it is important to note that floating-point numbers have **precision**, or how accurate the approximation of the number is. This usually relates to how accurate the fractional part is before it starts deviating from the correct value, causing error propagation to occur if you count those bits as significant. Therefore, when comparing equality between floats $f$ and $g$, it is generally more appropriate to write $abs(f - g) < \varepsilon$, for $\varepsilon = 0.000001$ or something similar, rather than to write $f == g$. The bottom line is, use the most memory-efficient data type that won't result in overflow errors. An example of a float is $1.05e - 7$, which is equal to 0.000000105. The $e$ signifies that the value is $1.05 \cdot 10^{-7}$.

| **Name** | **Size (bits)** | **Exponent Size (bits)** | **Significand size (bits)** | **Java** | **C++** |
|---|---|---|---|---|---|
| single-precision | 32 | 8 | 23 | `float` | `float` |
| double-precision | 64 | 11 | 52 | `double` | `double` |

## Recursion

Recursive functions are an important method of computation. The idea behind recursion is similar to loops: to repeat a process over and over again until a particular condition is satisfied. Then, the results are returned through the long line of recursions until the initial returns the final value of the recursion.

Recursive functions have a very definite structure, and if part of the structure is missing (especially the base case), the function will not work as intended:

```
function recursion(parameters){
        //- initializing of variables
        //- checking for base case(s)
        //- preprocessing
        //- search for appropriate parameters to recur on
        //- recursive step (call function with new parameters)
        //- return final value from the recursion
}
```

While the necessity of initializing and preprocessing is obvious, the step between them is critical but often overlooked. Base cases designate when the recursion should stop (see examples below), and without them, your recursive function will continue recurring even if the parameters make no logical sense. This will cause one of two errors: runtime errors due to out-of-bounds parameters, or a recursion depth error due to your machine running out of memory. Both of these are infinitely bad errors, so you should take great care to avoid them, and correct them if they appear.

Recursion is most useful for when you need to feed in outputs as inputs to the same function over and over again. Usually, recursion can be unraveled using queues (or stacks), but that generally takes immensely more memory, due to the costs of storing large queues of parameters. However, because recursive functions run on stack allocations, they generally have less RAM to work with, thus making it "easier," per se, to run out of memory from recursion than from storing parameters. Yet, recursion is still a simple and practical solution.

The most common example of recursion is the Fibonacci program. The Fibonacci numbers are, in fact, defined recursively, thus making them the perfect practice recursion program. Fibonacci numbers are defined as: $F_n = F_{n-1} + F_{n-2}$, for $n > 1$, with $F_0 = F_1 = 1$. From this definition, we know all we need to know in order to write the recursive function to calculate the $n^{th}$ Fibonacci number: the recursive step (definition of the Fibonacci numbers), the parameter conditions ($n > 1$), and the base cases ($F_0 = F_1 = 1$). Thus, the program is as follows:

| Java | C++ | Python |
|---|---|---|
| `public int Fib(int n){`<br>`  //no initializing step`<br>`  //no preprocessing step`<br>`  if(n<2){ //base case`<br>`  return 1;`<br>`  }`<br>`  //recursive step & return`<br>`  return Fib(n-1)+Fib(n-2);`<br>`}` | `public int Fib(int n){`<br>`  //no initializing step`<br>`  //no preprocessing step`<br>`  if(n<2){ //base case`<br>`  return 1;`<br>`  }`<br>`  //recursive step & return`<br>`  return Fib(n-1)+Fib(n-2);`<br>`}` | `def Fib(n):`<br>`  //no initializing step`<br>`  //no preprocessing step`<br>`  if n<2:  #base case`<br>`  return 1`<br>`  return Fib(n-1)+Fib(n-2)` |

As you will notice, some of the Fibonacci numbers will be calculated many times over, especially for very large $n$. This might lead to reduced efficiency and increased memory usage. There is a way to rectify this, but we will discuss that later in the Dynamic Programming section.

Sample Problem #1: Identify the error in the C++ code below for the Lucas numbers (similar to the Fibonacci numbers, except $F_0 = 2$ and $F_1 = 1$):

```
int Lucas(int n){
        return Lucas(n-1)+Lucas(n-2);
}
```

Answer: The error here is that there is no base case; this function will just keep running and running until the stack (not to be confused with the data structure) runs out of memory.

Sample Problem #2: Correct the code from Sample Problem #1, given the above definition of the Lucas numbers.

Answer: We simply insert the base case conditions, as there are no initialization or preprocessing requirements (changes are underlined):

```
int Lucas(int n){
        if(n==0)return 2;
        else if(n==1)return 1;
        return Lucas(n-1)+Lucas(n-2);
}
```

## Order of Operations

Many problems require basic implementation of things we normally do in math. For this, it is useful to know the operations that languages such as Java, C++, and Python have. The following is a table that might be handy. The order of operations is shown on the left.

| **Precedence (1=highest)** | **Java** | **C++** | **Python** | **Description** |
|---|---|---|---|---|
| 1 | () | () | () | parentheses |
| 2 | [method]() | [function]() | def() | function call |
| 3 | (type) | (type) | type() | type casting |
| 4 | | | `a**b` | $a^b$ |
| 5 | `a*b`,`a/b` | `a*b`,`a/b` | `a*b`,`a/b` | $a \cdot b$, $\frac{a}{b}$ |
| 6 | `a+b`,`a-b` | `a+b`,`a-b` | `a+b`,`a-b` | $a + b$, $a - b$ |

# Object-Oriented Programming

Object-Oriented Programming (OOP) is a style of programming that focuses on the creation, manipulation, and interaction of and between objects. Objects, then, are structures that represent pieces of data. As such, the most important objects are the standard data structures: various linked lists, stacks, queues, and trees. All of these are crucial components of any well-written object-oriented program, and you have probably already used these data structures in some way or another.

Each language has its own constructions of these data structures; some have various different implementations to suit different needs. Below is a table that lists the basic data structure classes available in Java and C++ (T simply means "type," E means "element"; both refer to generic data types). Note that Python, as an untyped language, can implement stacks, queues, and trees simply from lists. Also, C does not have built-in data structures.

| Structure | Java | C++ |
|---|---|---|
| linked list | `ArrayList<E>`, `Vector<E>`, `LinkedList<E>` | `vector<T>` |
| stack | `Stack<E>` | `stack<T>` |
| queue | `Queue<E>`, `PriorityQueue<E>` | `queue<T>`, `priority_queue<T>` |

In addition to these basic structures, there are also two slightly more complex data structures: maps and sets. Maps are like arrays and lists, except that the keys (analogous to indices) can be any type. Sets are also similar to arrays and lists, but they preclude duplicates from appearing in the set. (As above, Python has its own typeless maps and sets: dictionaries and sets. Both are implemented using curly braces {} and will not be covered here. `http://docs.python.org/tutorial/datastructures.html#sets` has some pretty good information about Python dictionaries and sets if you want to look at that.) Again, T and E mean the same as before, and K are V are obviously generic-type Keys and Values (respectively)

| Structure | Java | C++ |
|---|---|---|
| map | `TreeMap<K,V>`, `HashMap<K,V>` | `map<K,T>` |
| set | `TreeSet<E>`, `HashSet<E>` | `set<T>` |

As can be noticed with the Java data structures, there are two popular implementations of maps and sets: tree and hash implementations (C++'s map and set classes are both implemented with trees). The main differences between the two implementation types is are in their relative efficiencies in node manipulation (*e.g.* insertion and deletion) and node searching. The table below summarizes the key differences between tree and map implementations of maps and sets (in the average case).

| Operation | Tree efficiency | Hash efficiency | More efficient |
|---|---|---|---|
| Insertion | O($\log n$) | O(1) | Hash |
| Deletion | O($\log n$) | O(1) | Hash |
| Searching | O($\log n$) | O(1) | Hash |

Big-O notation will be covered in a later section. Alternatively, you are free to look it up online.

As you can see, hash-based maps and sets insert elements in constant time – that's really fast! On the other hand, it takes linear time to search for something, and while that should work most of the time, there are several instances where linear time and logarithmic time are significantly different. Looking at tree-based maps and sets, we can see that they insert elements in logarithmic time – not bad, but nothing compared to constant time (too bad). But, as noted above, tree-based implementations are significantly faster in searching, especially for large data sets. Thus, one common practice among programmers (especially when using Java) is to insert data into hash-based maps and sets, then recast them as tree-based maps and sets for data searching and processing. In this way, the only considerably-large part of the runtime occurs during the recasting.

In addition, Java, C++, and Python all have the capabilities to define custom objects known as **class**es. You should be familiar with how custom **class** constructs work in your preferred language; if you are not, please ask your peers or teacher(s) for help, as this will inevitably be part of the competition in some form.

We will, however, add some key points about classes. First, each language's classes have their own quirks: Java makes inheritance and subclasses extremely easy; C++ allows custom definitions of any operator, as well as destructor functions; and Python also allows [limited] custom definitions of operators, though it requires the use of self which occasionally results in interesting problems for function calls. Moreover, please note that scopes (`public`, `private`, and `protected`) are also commonly misused. Finally, it is generally good convention to use accessor and mutator functions (and methods) to access and manipulate the internal variables of an object's instances, rather than allowing direct access to these variables.

Also, one key thing to note about OOP is that OOP programs tend to use functions (and methods and defs) a lot. Each function serves a single purpose, and does not do much more than that purpose. Therefore, your main function should be reserved mainly for object creation and function calls. This really helps reduce the clutter in your program, making it easier to debug, especially when using classes and custom-defined objects. Of course, this is not evident in small programs, but large-scale projects would be extremely messy and difficult to fix if the program is cluttered.

# Code Theory

For some of these problems, you will be asked to analyze and work with code. In order to be fair, we will be offering different versions of the test for Java, C++, and Python 3. Other languages will be considered if we are given enough advanced notice. Please note that some problems might not be available in another language in order to avoid undermining the purpose of the problem. All versions of the test will have the same answers (that is, the only difference is the appearance of the code). Because the code will be available in all these languages, elements specific to one language only will not be used unless there is an equivalent in another language. Some language equivalents are shown in the table below:

| **Java** | **C++** | **Python 3** |
|---|---|---|
| ArrayList | vector | list: [1, 2, 3] |
| HashSet, TreeSet | set | set: {1, 2, 3}, set() not {} for the empty set |
| HashMap, TreeMap | map | dictionary: {1 : 2, 2: 3, 3 : 1} |
| `public class Pair{`<br>`  int a, b;`<br>`  public Pair(int x, int y){`<br>`  a = x; b = y;`<br>`  }`<br>`}` | pair | tuple: (1, 2) |

Note that as in the last example, we may define our own classes to make up for missing equivalents. However, these self-defined equivalents will never be complex. For example, we will not implement a HashMap in C++ as part of a problem.

Occasionally, a C++ `vector` might be used as an equivalent to a Java array. The purpose of this is to avoid dealing with pointers, which are rather messy and beyond the scope of this competition.

## Code Analysis

You will be given a snippet of a program and your job is to deduce what the program will output. These programs will not contain highly advanced syntactical features, but may contain complex arrangements of loops and if-statements that require some thinking to process. Often, a much simpler concept will be disguised in a complicated-looking program.

For all such problems, assume that all relevant import / include statements are present (but not shown). The code shown is guaranteed to compile and run without errors. That is, there will always be an answer printed.

Sample Problem #1:

| Java | C++ | Python 3 |
|---|---|---|
| `int N = 100;`<br>`int k = 0;`<br>`for(int i = 1; i < N; i++)`<br>`  if(N % i == 0)`<br>`  k++;`<br>`System.out.println(k);` | `int N = 100;`<br>`int k = 0;`<br>`for(int i = 1; i < N; i++)`<br>`  if(N % i == 0)`<br>`  k++;`<br>`cout << k << endl;` | `N = 100`<br>`k = 0`<br>`for i in range(1, N):`<br>`  if N % i == 0:`<br>`  k += 1`<br>`print(k)` |

In order to solve these problems, it is necessary to know basic syntax and some mathematical techniques. The program above loops through all values from 1 to N and increments k for all values that are a factor of N (the modulus operator, %, returns the remainder when the first argument is divided by the second argument). To interpret this more intuitively, k becomes the number of factors (not including N) of N. For N = 100, the relevant factors would be 1, 2, 4, 5, 10, 20, 25, and 50. At the end of the loop, k equals 8.

Answer: **8**

Sample Problem #2:

| Java | C++ | Python 3 |
|---|---|---|
| `ArrayList<Integer> a`<br>`  = new ArrayList<Integer>();`<br>`for(int i = 0; i < 100; i++)`<br>`  if(i * i > 1000)`<br>`  a.add(i * i);`<br>`System.out.println(a.size());` | `vector<int> a;`<br><br>`for(int i = 0; i < 100; i++)`<br>`  if(i * i > 1000)`<br>`  a.push_back(i * i);`<br>`cout << a.size() << endl;` | `vals = []`<br><br>`for i in range(0, 100):`<br>`  if i * i > 1000:`<br>`  vals.append(i * i)`<br>`print(len(vals))` |

Solving this problem requires knowledge of lists (ArrayLists in Java and vectors in C++). These lists are variable-size arrays: instead of having a set size like an array does, lists can append elements to the end and increase their size. The code snippet checks through values from 0 to 99 and adds its square to the list if the square is greater than 1000. In the end, the size of this list (the answer to this problem) is equal to the number of values from 0 to 99 whose square is greater than 1000. Noting that the squares of values from 0 ($0^2 = 0$) to 31 ($31^2 = 961$) are less than 1000, the squares of the values from 32 to 99 are all greater than 1000. The answer to this problem is the (99 - 32 + 1) = 68.

Answer: **68**

## Code Writing

Some problems will give you code intended for a specific purpose, but the code will be missing sections. It is your job to figure out what is missing in order to allow the code to fulfill its purpose. If there are multiple solutions, all will be accepted, though obfuscated code and

unnecessary code will be penalized. Additionally, you will not be allowed to insert code in spaces that are not missing code. Missing code should be inserted in the location marked by underlines (_____). Finally, be aware that some problems may require you to write entire code fragments (within context) such as classes and functions.

Once again, all code is assumed to compile and all necessary imports / includes are assumed to exist (but are not shown).

Sample Problem #1:

```
Glenna is writing a program to output all numbers less than 1000 that are both a multiple of
2 and a multiple of 3.  Help Glenna fill in the missing code.
```

| **Java** | **C++** | **Python 3** |
|---|---|---|
| `for(int i = 1; i < 1000; i++)`<br>`  if(_____)`<br>`  System.out.println(i);` | `for(int i = 1; i < 1000; i++)`<br>`  if(_____)`<br>`  cout << i << endl;` | `for i in range(1, 1001):`<br>`  if _____:`<br>`  print(i)` |

The missing code should return a boolean that checks whether a number fits the criteria. In this case, it would be true if i is divisible by both 2 and 3 and false otherwise. There are many possible solutions here, but the most common two are shown in the table below.

| **Java** | **C++** | **Python 3** |
|---|---|---|
| `i % 2 == 0 && i % 3 == 0` | `i % 2 == 0 && i % 3 == 0` | `i % 2 == 0 and i % 3 == 0` |
| `i % 6 == 0` | `i % 6 == 0` | `i % 6 == 0` |

The answers are expected to be syntactically correct (and conceptually correct). For example, substituting an "=" for an "==" in the answer above would result in a wrong answer.

## Code Debugging

Sometimes, code provided will intentionally include an error. It will be your job to find the error (and possibly fix it). This bug will always be a conceptual bug (*i.e.* a logic error) and not one that causes the program not to compile. That is, the code snippet given will always be a part of a program that both compiles and runs successfully, but there will be an error in the program that causes the output to be incorrect. Note that, due to their very nature, these problems might require a significant amount of knowledge of the objects (most likely data structures) involved.

Sample Problem #1:

```
Albert has written some code to reverse an input array.  However, his code does not work.
Circle the part of the code that is incorrect and tell Albert what he should have written
instead to make his reverse function work.
```

| Java | C++ | Python 3 |
|---|---|---|
| `public void reverse(int[] array){`<br>`  int n = array.length;`<br>`  for(int i = 0; i < n; i++){`<br>`  int temp = array[i];`<br>`  array[i] = array[n - i - 1];`<br>`  array[n - i - 1] = temp;`<br>`  }`<br>`}` | `void reverse(vector<int> array){`<br>`  int n = array.size();`<br>`  for(int i = 0; i < n; i++){`<br>`  int temp = array[i];`<br>`  array[i] = array[n - i - 1];`<br>`  array[n - i - 1] = temp;`<br>`  }`<br>`}` | `def reverse(array):`<br>`  n = len(array)`<br>`  for i in range(n):`<br>`  temp = array[i]`<br>`  array[i] = array[n - i - 1]`<br>`  array[n - i - 1] = temp` |

One way to solve this problem is to simply try a test case. What if the array inputted is $\{1, 2, 3, 4\}$? The result we want is $\{4, 3, 2, 1\}$, but a hand-run of the code above first switches the values in the first half of the array with values from the other side, to produce $\{4, 3, 2, 1\}$, but then proceeds to switch them all back to produce $\{1, 2, 3, 4\}$ again. This is not correct.

To properly reverse the array, the loop must only go through the first half of the array and swap those values with values in the second half. There are several ways to correct this code, but one way is shown below (the changes to the code are boxed):

| Java | C++ | Python 3 |
|---|---|---|
| `public void reverse(int[] array){`<br>`  int n = array.length;`<br>`  for(int i = 0; i < n/2; i++){`<br>`  int temp = array[i];`<br>`  array[i] = array[n - i - 1];`<br>`  array[n - i - 1] = temp;`<br>`  }`<br>`}` | `void reverse(vector<int> array){`<br>`  int n = array.size();`<br>`  for(int i = 0; i < n/2; i++){`<br>`  int temp = array[i];`<br>`  array[i] = array[n - i - 1];`<br>`  array[n - i - 1] = temp;`<br>`  }`<br>`}` | `def reverse(array):`<br>`  n = len(array)`<br>`  for i in range(int(n/2)):`<br>`  temp = array[i]`<br>`  array[i] = array[n - i - 1]`<br>`  array[n - i - 1] = temp` |

# Boolean Algebra

Boolean Algebra is a form of algebra in which there are only two possible values: `true` and `false`, represented with the Boolean data type. This form of algebra is useful mostly for `if` statements in programming. In Boolean Algebra, there are operations just as with regular algebra. However, these operations are different.

## Operators

The first operation is the AND operation. Suppose $A$ and $B$ are two Boolean variables (that is, $A$ is either `true` or `false`, and $B$ is also either `true` or `false`). $A$ AND $B$ evaluates to `true` if and only if both $A$ and $B$ are `true`. Else, it evaluates to `false`. This can be illustrated with logic. If I have two `true` statements, such as "the year has 12 months" and "Java is a programming language," then the sentence "the year has 12 months and Java is a programming language" is `true`. However, changing either of them to a `false` statement results in something `false`: "the year has 11 months and Java is a programming language."

The second operation is the OR operation. $A$ OR $B$ evaluates to `true` as long as at least one of $A$ and $B$ is `true`. If both $A$ and $B$ are `false`, then it evaluates to `false`. As an example, "the year has 11 months or Java is a programming language" is `true` but "the year as 11 months or ants are mammals" is `false`.

The third operation is the NOT operation. It is a unary operation, unlike OR and AND, which are binary operations (take two operands). NOT $A$ simply reverses $A$. If $A$ is `true`, then NOT $A$ is `false`. If $A$ is `false`, then NOT $A$ is `true`.

The fourth operation is the XOR operation. It is a binary operation for which $A$ XOR $B$ returns `true` if and only if $A$ and B have different values. XOR stands for "exclusive-or" because XOR is the same as OR with the exception that `true` XOR `true` is `false`.

In Boolean Algebra notation, $+$ is OR and $\bullet$ is AND. XOR is sometimes a plus with a circle around it. With this notation, common algebraic simplification such as the distributive property is valid. NOT is sometimes represented with an overline: $\overline{A}$.

Boolean Algebra has its own order of operations. Parentheses have the highest precedence, followed by NOT, then AND (evaluated from left to right), XOR (left to right), and then OR (left to right).

## Truth Tables

Truth tables are a way to prove that two Boolean statements are equal. For an example, lets try to prove that $A$ OR $A$ AND $B$ is always the same as $A$. In order to prove this, we must consider the value of both statements for every possible pair of values $(A, B)$. Note that often, we let 0 represent `false` and 1 represent `true`.

| $A$ | $B$ | $A$ AND $B$ | $A$ OR $A$ AND $B$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Note how this is obtained. Using the pattern for the first two columns, each row expresses the results for one possible pair of values $(A, B)$. The third column is obtained by using the AND operation on the first two columns. The first column OR the third column gives the fourth column. Because the first column ($A$) equals the fourth column ($A$ OR $A$ AND $B$) in every value, we know that $A \text{ OR } A \text{ AND } B = A$.

## Identities

With Boolean Algebra, it is useful to know some identities. Every identity below can be proved using algebraic manipulation or through truth tables.

1. $A + 0 = A$ (OR identity)
2. $A + 1 = 1$
3. $A + A = A$
4. $\overline{\overline{A}} = A$
5. $A \bullet 0 = 0$
6. $A \bullet 1 = A$ (AND identity)
7. $A \bullet B = B \bullet A$, $A + B = B + A$ (commutative property)
8. $(A \bullet B) \bullet C = A \bullet (B \bullet C)$ (associative property)
9. $(A + B) + C = A + (B + C)$ (associative property)
10. $A + B \bullet C = (A + B) \bullet (A + C)$ (distributive property)
11. $A \bullet (B + C) = (A \bullet B) + (A \bullet C)$ (distributive property)
12. $A + (A \bullet B) = A$
13. $\overline{(X \bullet Y)} = \overline{X} + \overline{Y}$ (DeMorgan's Law)
14. $\overline{(X + Y)} = \overline{X} \bullet \overline{Y}$ (DeMorgan's Law)

We proved #11 above using a truth table, but we can also prove it algebraically.

$$A + (A \bullet B) = (A \bullet 1) + (A \bullet B) = A \bullet (1 + B) = A \bullet 1 = A$$

# Code Efficiency†

When designing code, especially if it must endure extreme loads, it is important to consider how long the program takes to execute, the program's *performance*, and how much of the computer's memory the program uses during a run, the program's *space complexity*. Programs that are too slow, in the contest setting, will exceed the time limit and, in the real world, can slow down people and other programs dependent on it and can be a nuisance to those who use them. Programs that use too much memory can be rejected in contests just as slow programs can be. Real world programs with memory problems can cause the computer or network the program is run on to freeze and can fail if more memory is needed than is available. The goal of efficiency analyis is to determine which algorithms run the fastest and with the least memory by quantifying an algorithm's performance and space complexity.

## Big O Notation

Unsurprisingly, a program's execution time is directly proportional to the number of operations it must perform. In other words the more "stuff" the program has to do, the longer it takes. We can write down a formula for the number of operations a given algorithm uses. Hypothetically, say you have an algorithm that multiplies two numbers and then, given an array of numbers of length N, multiplies each number in the array by 1, then by 2, and then by 3. This algorithm requires 3N + 1 operations, because it takes 1 operation to multiply the first two numbers and N operations to go through each number in the array and multiply it by another number, which must be done three times. To express this, we use what is known as *big O notation* is used. To indicate that the operation takes 3N + 1 operations, we say that is has performance O(3N + 1) (read "oh of three-n plus one").

Computer scientists are often more concerned with how a program performs in the long run than they are in getting an exact formula. As N becomes larger in the example above, the 1 in 3N + 1 becomes increasing insignificant. Also, we drop the coefficients of the terms (if they are constant), in this case the 3 in 3N + 1. Why? Big O notation attempts to describe what happens to the running time of an algorithm as the data increases. If N doubles in size, an algorithm with performance O(N) will double the time it takes. An algorithm of O(3N) has this same behavior, so it is simplified to O(N). Hence, to indicate an algorithm does not change its performance at all based on input (it takes a constant amount of time no matter what), that algorithm has a performance of O(1). So, while we said earlier that the performance of our pretend algorithm was O(3N + 1), it more properly has a performance O(N).

Suppose that some mathematical genius were to look at your algorithm and point out that instead of multiplying each number in the array by 1 and then by 2 and then by 3, we could simply multiply each number by 6 and only loop over the array once. What is the performance of the new and improved algorithm in big O notation? Well, now we only require N + 1 operations instead of 3N + 1, so the algorithm has a performance O(N). Note that the

first algorithm, despite being three times slower, had the same big O value for performance. This is one of the pit falls of big O notation, and when evaluating algorithms for contests, it may be helpful to include coefficients and such in your calculation so you know the total number of operations performed. The rule of thumb is that the computer can perform on the order of 100,000 operations per second.

If we wanted to calculate space complexity, we would apply the same logic but towards memory rather than operations. Through we may need some memory to hold some of the other numbers, the biggest spender is the array which requires N slots of memory. Therefore, memory grows with N, and the algorithm has a space complexity of O(N).

## Determining Big O

To find the big O of any algorithm, add up the big O values of operations that happen sequentially and multiply by the number of times you loop over it. For example, say we have the following pseudocode:

```
for i from 0 to N/2:
  do an O(N) operation
  do an O(N^2) operation
```

The entire algorithm has a performance of $O(\frac{N}{2}(N+N^2))$ which, because $N^2$ is larger than N and $O(\frac{N}{2})=O(N)$, simplifies to just $O(N^3)$.

## Common Big O Values

The most common big O values are summarized in the table along with some examples of algorithms with that efficiency. If you don't recognize some of the algorithms, that is okay. The key is to be able to apply the ideas behind big O to problems and towards the programs you write.

| Value | Name | Examples |
|---|---|---|
| $O(1)$ | Constant | Determining even or odd, Array retrieval |
| $O(\log N)$ | Logarithmic | Binary search, Heap deletion and addition |
| $O(N)$ | Linear | Linear search, Array summation |
| $O(N \log N)$ | Linearithmic | Quicksort, Heap building |
| $O(N^2)$ | Quadratic | Selection sort, Worst-case quicksort |
| $O(N^3)$ | Cubic | Floyd-Warshall algorithm |
| $O(2^N)$ | Exponential | Naive Fibonacci number calculation |

Note that while the base of the logarithm does not matter when determining the big O value, but due to the many operations in computer science that split the data in two, such as a binary search, the logarithm is usually considered to have a base of 2, but it could have a base of any number. Likewise, the base of the exponential function can be any value as well.

# Data Structures†

Basic data structures and problems related to these structures may be present on the test. Data structures to know and understand are:

- linked lists
- stacks
- queues
- trees
- binary search trees
- heaps

## Linear Data Structures (Lists, Stacks, and Queues)

A **linked list** is made up of a series of "nodes." Each node holds a value, and a link to the next node. Thus, we can walk through all stored values one at a time by following the chain of links.

The main disadvantage of linked lists is the inability to directly access a certain position in the list. For example, if we want to see the $10^{\text{th}}$ value, we would have to pass through the first 9 values in the list. This means that if we want to see the very last value, we would have to go through the entire list!

However, linked lists do have their advantages. Once we're looking at the node we want to work with, certain operations can be performed very easily. For example, suppose nodes $j$ and $k$ are linked together, and we want to put node $l$ between them. We can simply link $j$ to $l$ and $l$ to $k$, regardless of what nodes come before $j$ or after $k$. Similarly, deletion of a node only involes changing the link for the node directly before it. For example, suppose $m$ points to $n$ and $n$ points to $p$. We can delete node $n$ by simply setting $m$ to link directly to $p$ (thereby destroying the connection from $m$ to $n$).

So, linked lists can be very useful in situations where we're dealing with the data in order, and we want to perform insertion or deletion options. However, they can become a serious computational drain when we want to access specific positions in the data.

Linked lists are often modified to create **doubly-linked lists**, where each node also has a link to the *previous* node. Depending on the way you need to work with your data, this can be a time-saving enhancement.

A **stack** is a list of values that supports two operations: push and pop. Items can be pushed, or inserted, into a stack, and items can be popped, or removed, from the stack. A stack is

special because the value that is popped is always the one most recently added. A real-life example of a stack is a stack of dishes: when washing dishes, we always remove the dishes that were most recently added to the stack, because those dishes are on top.

A **queue** is also a list of values that supports the push and pop operations. A queue differs from a stack in that the item popped is the one that was first added to the queue. A real-life example of a queue is a line. When people wait in line, people enter at the end of the line and people leave from the front of the line. The earlier you enter the line, the earlier you can expect to leave.

Sample problem #1:

```
Albert has a stack.  He pushes 1, then 2, then 3, then 4.  He then pops two elements.  What
is the next item that Albert would pop?
```

After adding the four elements, the stack looks like: {1, 2, 3, 4}, with the elements being pushed and popped from the right side. The first pop operation pops 4, and the next one pops 2. The next item that Albert would pop is 2.

Answer: **2**

Sample Problem #2:

```
Albert has a queue.  He pushes 1, then 2, then 3, then 4.  He then pops two elements.  What
is the next item that Albert would pop?
```

After adding the four elements, the queue looks like: {1, 2, 3, 4}. Elements are pushed in on the right side but are popped from the left side. The first pop operation pops 1, and the next one pops 2. The next item that Albert would pop is 3.

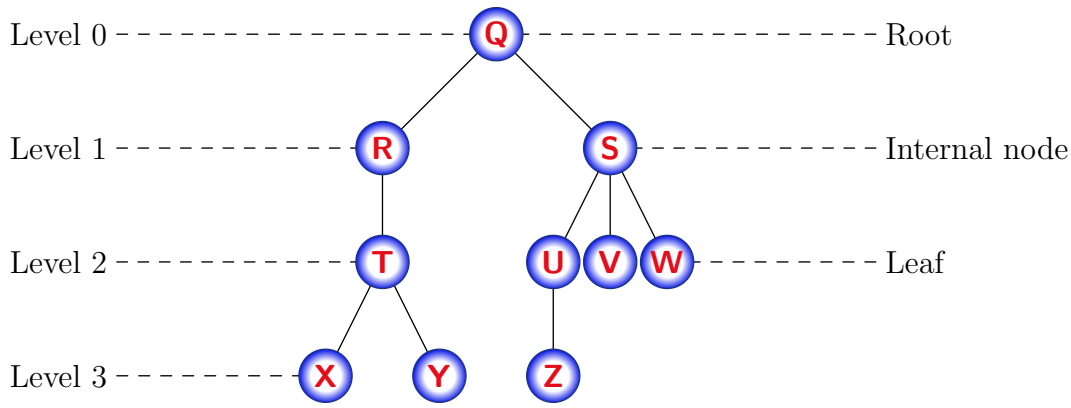
Answer: **3**

## Tree-Based Data Structures

A **tree** is another type of data structure based on nodes linking to other nodes. Think of a family tree showing all of the generations originating from one common ancestor. In the tree data structure, each node is linked to its parent, just as children on a family tree have a line drawn to their parents. Unlike a linked list, which can be considered to be one specific example of a tree (can you see why?), each node in a tree can have multiple children.

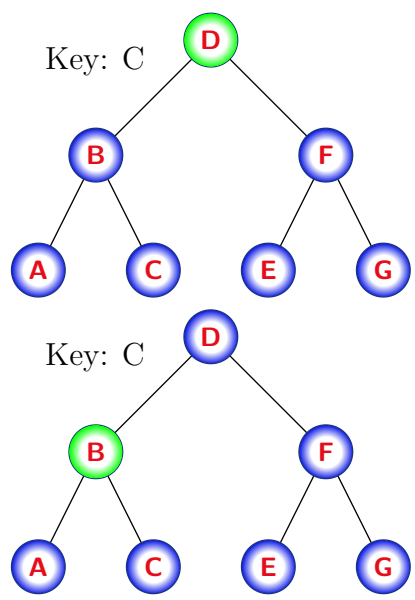
The node at the very top of the tree is the only one without a parent. This node is called the **root node**. Any nodes without children are referred to as **leaves**.

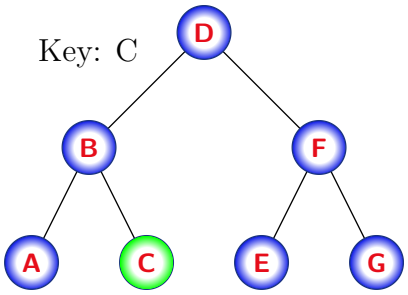
Below is a graphical representation of a generic tree. As always, each node stores one value.



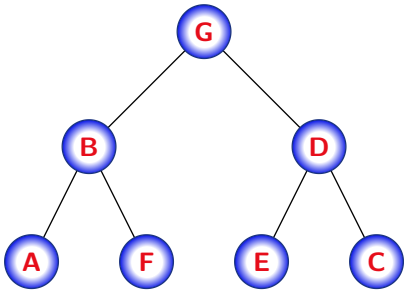

A **binary tree** is a special type of tree in which each node is only allowed to have up to 2 children. As we'll see below, binary trees are often used as the foundation for other data structures, because of their special properties.

A **binary search tree (sometimes BST)** is a particular kind of binary tree, with some restrictions on the values stored in the nodes. For any given node, the value in its left child, and all of the values in its left child's descendants, must be smaller than its own value. Similarly, all the value in its right child, and all of the values in its right child's descendants, must be greater than its own value. Binary trees are often used when we want to search through a data set for a specific value. Starting at the very top, we can compare the value in the current node to the value we're looking for, and know whether to go right or left. Every time we do this, we cut the size of the data set we need to look at **in half**. Thus, we can search through large data sets very quickly once they're stored as a binary tree.






A **heap** is another type of tree that places restrictions on the values stored in its nodes. In a *max-heap*, every node's value must be smaller than that of its parent. Thus, the maximum value will always be at the very top. In a *min-heap*, each node must have a value greater than that of its parent, meaning that the root node will always contain the minimum value.



# Graph Theory†

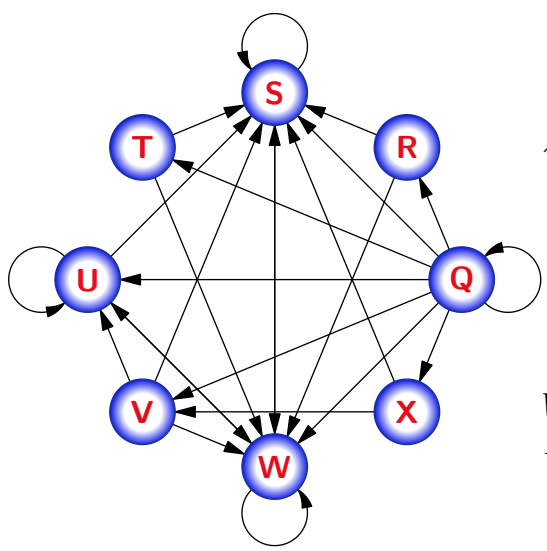
Graph theory relates to the purposes and uses of graphs in programs. Before we dive into graph *theory*, let's first quickly review what graphs *are*.

A graph is a way to model relations between objects. Each object is called a **node** or **vertex**, while each relation is called an **edge**. Therefore, in a graph, nodes are connected by edges. A **path** is way to traverse the graph along nodes and edges to get from one node to another. Of course, this is not always possible, as graphs are not always connected.

There are many terms that relate to graphs, and some of them will be covered here. A graph is **connected** if there is a path between every pair of nodes. A **connected component** of a graph is a subset of the nodes that forms a connected graph. Edges can have **weights** and **directions**. A weight of an edge can be correlated to the length of the edge or perhaps the time taken to traverse the edge. An **unweighted graph** has every edge weight equal to 1 (or otherwise uniform). Sometimes, edges can only be traversed in one direction. If an edge can be traversed in any direction, then it is **bidirectional**. A **directed graph** contains only directed edges, and an **undirected graph** contains undirected edges.

A **cycle** of a graph is any path that starts and ends in the same place. A **simple path** is a path that does not visit any node more than once. An **acyclic graph** contains no cycles. A graph is **sparse** if there are few edges and **dense** if there are many edges. A graph that contains every possible edge is a **complete graph**.

When storing a graph within a program, there are two main methods to consider for storing the edges. The first method is an adjacency matrix. For a graph with N nodes, the adjacency matrix is an N by N graph. The entry in row i and column j is 1 if there is an edge from node i to node j and 0 otherwise. The matrix is symmetric (about the main diagonal) if the graph is undirected. A sample graph and its associated adjacency matrix is below.




| $\nearrow$ | $Q$ | $R$ | $S$ | $T$ | $U$ | $V$ | $W$ | $X$ |
|---|---|---|---|---|---|---|---|---|
| $Q$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $R$ | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| $S$ | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| $T$ | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| $U$ | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| $V$ | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| $W$ | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| $X$ | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

The other way to store edges is with an adjacency list. For each node, an ArrayList (or vector or list) of edges originating from that node is stored. Adjacency lists, where each node is associated with a list of its neighbors and the distances between them, generally use less memory, but for dense graphs, function about the same as an adjacency matrix. For sparse graphs, adjacency lists are generally much faster.

Graph theory, then, uses a graph as the basis for modeling various situations. In fact, the most common use for graphs is searching, whether for a particular node or path. Over the decades, several different techniques for analyzing graphs have been developed based on various searching algorithms. The most common such techniques are flood fill and Dijkstra's algorithm.

Using flood fill, you start from one node of a graph and check each neighbor recursively until you find the node that you want or until you search past the bounds of your graph (thus implying the absence of the node you were searching for). Additionally, in order to prevent infinite recursion, a "visited" tag is required. Usually, flood fill searches are performed on grids, such as an $m$ by $n$ matrix. Therefore, flood fill requires $m \cdot n$ memory. Pseudo-code for this is below:

```
visited := m by n matrix
graph := m by n matrix of nodes
function flood(c):  //c is the current node
        if heuristic(c) == target value:  //base case
                return c;
        else:
                mark this node as visited;
        t := null; //target node
        for each neighbor n of c:
                if n not visited:
                        t := flood(n);
                        if t != null:  //found target node
                                break;
        return t;
```

As you can see, this is an $O(m \cdot n)$ program. That is, in the worst case, the program will go through all $m \cdot n$ nodes. This can be highly inefficient, especially for very large $m$ and $n$. However, it is still very useful for small-sized matrices. Note that flood fill can return a path as well, if the base case returned the node as a single-element array and if the return statement at the bottom had one line before it:

```
t.append(c);
```

Also, if we are looking for the shortest length of the path, rather than the path itself, then

the return value would be the best heuristic value (the total, for the base case), while each call to the flood fill would have an additional parameter: the heuristic value of the current path.

However, pathfinding using flood fill is laborious and inefficient. Moreover, it is very difficult to find the heuristically-shortest path between two nodes with flood fill. As such, a standard technique for pathfinding is Dijkstra's algorithm (often shortened to just "Dijkstra"). Dijkstra allows us to traverse a graph finding the shortest path quickly and efficiently. In particular, Dijkstra uses a priority queue (prioritizing on the heuristic value of a path) and the breadth-first search (BFS) technique. The pseudocode is given below:

```
function Dijkstra(graph, from, to):
    //graph[a][b] gives the heuristic value from a to b, if a and b.
    // this value is set to infinity if there is no path from a to b.
    Q := [priority] queue;
    visited := array of false values for each node;
    Q.add( array(0, from, array(from) ) ); //heuristic from->from is 0
    visited[from] := 1; //or true.  0 represents false.
    while Q is not empty:
        top := Q.top(); //node with the best heuristic value in Q
        Q.pop(); //remove current node
        h := top[0]; //heuristic of current node
        c := top[1]; //current node
        p := top[2]; //current node's path from from
        if c is to:
            return p;
        for each unvisited neighbor n of c:
            Q.add( array(h + graph[c][n], n, p.append(n) ) );
        sort(Q); //the path with the best heuristic value is first
    return empty path; //no path from from to to exists
```

What Dijkstra's algorithm says is: start at the root node, create paths from the root to each of its neighbors, put these paths in a queue, choose the path with the smallest heuristic value, have the last node in the path become the new root for that loop, and repeat the process, avoiding previously visited nodes. This guarantees that each path from the original root (also called source) to another node is the shortest possible. Therefore, if the new root is our target, its path from the source is guaranteed to be the shortest path from the source to the target, thus we return that path.

If we wanted the length of the shortest path from the source to the target, we can simplify Dijkstra to account for only the shortest length by removing the paths from the queued elements, and returning the heuristic of the target node instead of the path from the source.

# Search Techniques†

## Binary Search

Binary search is a very efficient searching algorithm. The precondition is that the array or list is heuristically presorted with "less" being at the front of the array or list and "greater" heuristic values being at the end.

The concept of a binary search is that the array can be thought of as a tree. Starting in the middle of the array, we traverse the imaginary tree by contracting the starting and ending indices until they either converge on top of one another (element not found) or they converge onto one element (element found). The key here is that, for each iteration, the endpoints are contracted so that the resulting array or list is about half the length of the original one. This may sound confusing, so hopefully the pseudocode below will clear things up:

```
//returns true if key is found, and false if not
function binarySearch(array graph, element search):
        start := 0
        end := graph's length
        while end - start > 1:
                if heuristic(graph[(start+end)/2]) < heuristic(search):
                        end = (start+end)/2
                else:
                        start = (start+end)/2
        if end - start == 1:
                return true
        else:
                return false
```

Because this search method reduces the size of the searching array by around half, it is called a binary search, binary in this case referring to 2. With every iteration, the range of the array to search through is cut in half. An array of length $N$ can be cut in half $log_2 N$ times before it has size 1. Thus, the efficiency of this algorithm is $O(\log N)$, where $N$ is the number of elements to search through. Again, this is only feasible if the array is presorted on the heuristic values of the elements in the array.

## Depth First Search

Depth First Search or DFS is a method of traversing every element in a tree (or graph, if you keep track of traversed nodes) through recursion. The algorithm is to start at the root, then keep track of both children and go to the first child, treating it as a new root, then repeating the process until we reach a leaf. Then, we'll go back one step and go to the second child of the parent of the leaf and continue traversal. This process is recursive and continues until

all nodes have been traversed.

### Implementation with Recursion

The pseudocode is fairly simple. Suppose we simply want to traverse the nodes. Each node knows its children (in a list `node.children`). The following is pseudocode for a generic DFS implementation (try tracing the path the program takes on the diagram below):

```
call DFS(root)
procedure DFS(node):  // DFS function
       if node.value == goal:  do_something() // Check if we found the correct node
       for child in node.children:  // Recursively visit each child
              call DFS(child) // This is the recursive statement
       return
```

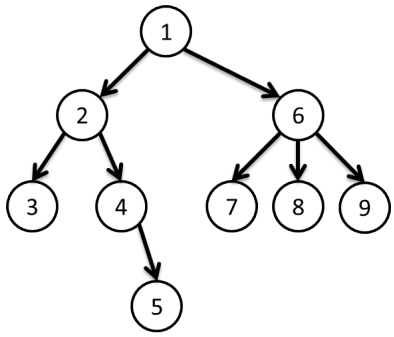



Figure 1: Node Traversal With Depth First Search

## Breadth First Search

If you noticed, DFS traverses down the tree, then across. Alternately, you can traverse the tree layer by layer using Breadth First Search (BFS). To do this, initialize a list with just the root node. Then iteratively, for each node in the list, add its children to the end of the list.

### Implementation with Queues

To implement this algorithm, we will use a queue, which was covered in the Data Structures section. Pseudocode for traversing a tree in a BFS manner might look like this:

```
queue.push(root)
while queue not empty:
       node := queue.pop() // Take a node from the front of the queue
       if node.value == goal:  do_something() // Check if we found the correct node
       for child in node.children:  // Push each child of the current node
```

```
queue.push(child) // to the back of the queue
```

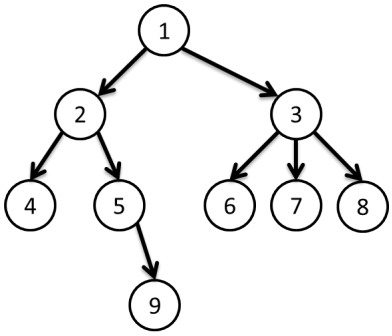



Figure 2: Node Traversal With Breadth First Search

**Challenge:** Can you modify it to work for any undirected, connected graph?

## DFS or BFS?

It's very important to know when to use BFS or DFS. While both methods will eventually find a solution, chosing the right method can drastically reduce runtime. BFS is good for finding the desired node closest to the root node, while DFS is useful for trees with a small height (because of memory usage). If you have a binary tree and want to find if any element equals 2, should you use BFS or DFS? What should you use to find the closest element to some start node that solves a problem?

# Dynamic Programming†

## What is Dynamic Programming?

Dynamic Programming is the misleading name given to a general method of solving certain optimization problems. Chances are if a problem asks you to optimize something and doesn't involve a graph, you'll want to use Dynamic Programming. As always, there are exceptions (for example, for problems with small upper bounds on $N$, brute force might actually be the way to go).

Dynamic Programming works on problems that can be represented as a series of sub-states. We can solve the smallest or base state first, then work up from there building up to the solution. Since we only calculate each sub-state once, the runtime of dynamic programming solutions is polynomial.

### Simple Example

A simple and overused example of dynamic programming is calculation of the Fibonacci numbers. Calculating the Fibonacci numbers recursively has an exponential time complexity $O(\varphi^N)$. But if we work from the bottom up calculating $F_2$, then $F_3$, etc, it's clear that this is $O(N)$.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 |

Figure 3: DP Array for Calculation of the First Few Fibonacci Numbers

### Simple Example Extension - Sliding Windows

A lot of memory is wasted in keeping old Fibonacci values that are no longer used. In fact, we really only need to store three numbers at any given time (or perhaps fewer if you're tricky). Keeping only the necessary information is known as ***sliding window***, and you may find it critical for some programming problems.

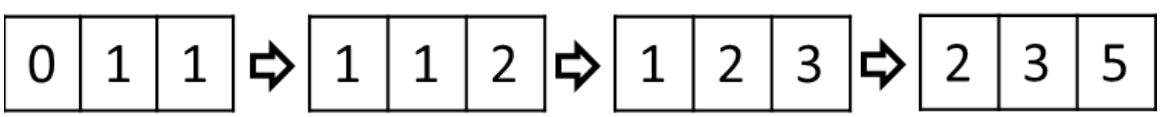



Figure 4: Calculation of Fibonacci Numbers Using DP and Sliding Window

## Knapsack Algorithm

One very common DP problem you'll see on many programming contests is the Knapsack Problem. There are various forms of the Knapsack problem, but the general idea is that

you have a "knapsack" with some capacity $C$. You have to fill it with any number of $N$ types of objects of some weight $W[i]$ and value $V[i]$. Given that the sum of the weights must not exceed $C$, find the maximum value that can be stored in the knapsack. This specific form, where you have an infinite number of discrete objects, is often known as the Integer Knapsack Problem. By way of example, consider the following case:

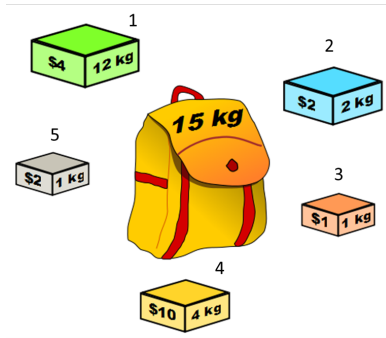



Figure 5: Example Knapsack Problem

### Brute Force

We know that each of the $N$ objects is *in* the knapsack some number of times. We could iterate over each object over the number of times it could be in the knapsack, but this is clearly too slow - exponential in $N$.

### The *Wrong* Way

You might think that we can just greedily add the objects with the highest value to weight first. But consider the case $C = 10$ and we have two objects: $\{V_1 = 8, W_1 = 6\}$, $\{V_2 = 5, W_2 = 5\}$. We would greedily add object 1, yielding a value of 8. However, a cleverer person would add two object 2s, thus yielding a value of 10.

### Dynamic Programming

Suppose instead of finding the maximum value obtainable with $C$ as the capacity, we found the maximum value for some lower capacity limit. Finding the maximum for $C' = 0$, for example, is trivial. If we now take $C' = 1$, we can just find which objects have a weight of 1 and maximize over their values. In fact, as we increment $C'$ we begin to notice a general pattern. Let's construct an array `dp[i]` that denotes the max value of a knapsack with capacity $i$. Then with a bit of thought, it's not hard to realize $dp[i] = \max_{\text{item } j} [dp[i - W[j]] + V[j]]$, assuming we initialize `dp[1...C]` to $-\infty$. Take a moment to understand why this works.

This function checks whether adding item $j$ is beneficial or not. If you choose to add item $j$, the value will be $dp[i - W[j]] + V[j]$, since this is the sum of the value of a knapsack of capacity $C -$ Weight of item $j$, and the value of item $j$. Then, we simply want the max

possible value over all possible item $j$'s to add.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 10 | 12 | 13 | 16 | 20 | 22 | 24 | 26 | 30 | 32 | 34 | 36 |

Figure 6: DP Array - Top Row Represents Capacity, Bottom Row Represents Optimum Value

## General Strategy

Now that we've seen an example problem and its dynamic programming solution, is there a general strategy for solving DP problems? In fact there is.

### Definition of State Variables

In this step, we essentially need to figure out what the substates are and what variables will change for smaller or larger substates. We can then construct some array `dp` where accessing `dp[i][j][k]...` returns the desired answer for a given subproblem of those variables $i, j, k, \ldots$.

In the Knapsack example, our substates were the maximum values of the knapsack for smaller capacities. We could then base the substate on the capacity, creating a one-dimensional `dp` array. For some substate capacity $i$, `dp[i]` returned the maximum attainable value for that capacity.

### Base Cases

Having figured out the state variables, we need to determine some base cases, usually at points where the state variables are very small.

In the Knapsack example, we might want to initialize `dp[0]` to 0 (since a knapsack of 0 capacity holds 0 value). We might also want to initialize the other elements of the array to $-\infty$ so that when we maximize, we make sure not to choose an unattainable capacity. If we use this method, we have to be careful when choosing the "maximum" that `dp[C]` is actually a positive number. Instead, we might choose the highest element of `dp` that is positive, since this will clearly be the maximum possible value of the knapsack of capacity $C$.

### Transition Functions

Here's the part that usually requires lots of thought. We need to figure out the relationship between the elements in our array, and we need to be able to calculate them in a bottom-up approach so that our run time is polynomial.

In the Knapsack example, we chose the function $dp[i] = \max_{\text{item } j} [dp[i - W[j]] + V[j]]$. From this, we start at `dp[1]` and continue to `dp[C]`, making sure to traverse in that order (bottom-up). What made us choose this particular function is that we know we want to consider all the possible ways to get `dp[i]`, and we want to maximize its value. So for each object $j$, we simply check if the value of that object added to a knapsack of capacity $i - W[j]$ is greater than `dp[i]`.

# Changelogs

## Changelog 2.012.05

- Schedule in study materials was updated to match revised schedule. Notably:
  - Part I (Theoretical) scaled down to 60 minutes before lunch.
  - Part II (Practical) scaled up to 150 minutes after lunch.
  - Systems orientation is now after lunch, right before Part II.

## Changelog 2.012.04

- A minor typographical error was fixed in subsection "Common Big O Values."
- Two minor typographical errors were fixed in section "The TJ IOI Contest."
- Minor typographical changes were made to section "Preface."
- A minor change was made in the solution to Sample Problem 1 for "Recursion," as well as in the pseudo-code.

## Changelog 2.012.03

- In subsection "Computer Number Formats," the explanation for sample problem 2 was updated.
- Minor language changes were added in subsection "Data Types." An example for a float was also added.
- Minor language changes were added in subsection "Recursion."
- Change in table in subsection "Order of Operations."
- In section "Object-Oriented Programming," we changed efficiency for searching in a hashing-based data structure to reflect average case efficiency.
- Minor fixes were made to the identities in "Boolean Algebra" section.
- Minor language changes were added in subsections "Big O Notation" and "Determining Big O."
- Minor language changes were made in section "Data Structures."
- Minor fixes to pseudocode and language in subsection "Binary Search."
- The word "dense" was bolded in section "Graph Theory."
- An expanded explanation was added for the dynamic programming transition function in subsection "Knapsack Algorithm."

## Changelog 2.012.02

- Pages in the Table of Contents are now numbered using capital Roman numerals: I, II, etc.
- In subsection "Data Types", a new paragraph has been added to show the steps of converting between integer powers of bases.
- In subsection "Data Types", a section about *comparing* floats floats has been added.
- Tables in the "Data Types" subsection have been updated to accurately reflect sizes.
  - Integer-based types table's values for integers and longs have been corrected to 4 and 8 bytes, respectively.
  - Floating-point-based types table's units have been changed from bytes to bits, which is more fitting.
- In subsection "Recursion", the structure has been updated to provide more clarity as to where the recursive step belongs.