4

Algorithms for Game Development

In chapter, the following recipes will be covered:

* Using sorting techniques to arrange items
* Using searching techniques to look for an item
* Finding the complexity of an algorithm
* Finding the endian-ness of a platform
* Using dynamic programming to break down a complex problem
* Using greedy algorithms to solve problems
* Using divide and conquer algorithms to solve problems

# Introduction

Algorithm refers to list of steps that should be applied to perform a task. Searching and Sorting algorithms are techniques by which we can search or sort elements in a container. A container by itself will have no advantage unless we can search or search items within that container. Based on certain containers, certain algorithms become more powerful on some than others. As an algorithm will run slower on a slower system and faster on a superior system, computation time is not an effective way to measure the effectiveness of an algorithm. Algorithms are rather measured as steps. Games are real time application. Hence algorithms that will be applied have to be effective for games to be executed at least at 30 frames per second. The ideal frame rate is 60 frames per second

# Using sorting techniques to arrange items

1. Sorting is a way to arrange items in a container. We can arrange them in ascending or ascending other. If we have to implement the high score system of a game and leader board, sorting becomes necessary. In the game, the moment a user achieves a sort higher than his previous highest score, we should update that value as the current highest score and push it to a local or an online leader board. If it’s local, we should arrange all the users’ previous high scores in descending order and display the top 10 score. If it is an online leader board, we need to sort all the users’ latest high scores and display the result.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe, we will find out how easy it is to arrange items in a container using different sorting techniques.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a header file called Sorting.h
5. Add the following lines of code.

**Sorting.h**

// Bubble Sort

template <class T>

void bubble\_sort(T a[], int n)

{

T temp;

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

{

for (int j = 0; j<n - i - 1; j++)

{

if (a[j]>a[j + 1])

{

temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

}

//Quick Sort

template <class T>

int partition(T a[], int p, int r)

{

T x;

int i;

x = a[r];

i = p - 1;

for (int j = p; j <= r - 1; j++)

{

if (a[j] <= x)

{

i = i + 1;

swap(a[i], a[j]);

}

}

swap(a[i + 1], a[r]);

return i + 1;

}

template <class T>

void quick\_sort(T a[], int p, int r)

{

int q;

if (p<r)

{

q = partition(a, p, r);

quick\_sort(a, p, q - 1);

quick\_sort(a, q + 1, r);

}

}

## How it works...

In this example bubble sort and quick sort have been discussed.

In games programming, we will be faced with numerous occasions when we need to sort data in a certain manner. However as easy as this may seem, we need to be careful with selection the correct sorting algorithm. Let us take the bubble sort algorithm. The algorithm is very simple as we use a nested loop to check for the sorting condition. However as you may notice that the inner conditional check will be executed n^2 number of times. Hence the order of the algorithm is O (n^2). If we are implementing a game like Candy Crush and we need to notify the user at run time if he has achieved a High Score and how many of his Facebook friend’s score he has crossed and what is the next score to beat, it might take an immense about of time. So on a large data set, it is not the most effective algorithm to use.

On the other hand, using quick sort does the trick for us. As you can see if divides the problem space continuously and reduces the search space. It does so by selecting a pivot and shifting the pivot at every step. This is very effective and is the most preferred sorting algorithm. Even the inbuilt standard template library uses a modified version of quick sort called QSort2 for its inbuilt sort () function.

# Using searching techniques to look for an item

Searching techniques are the group of algorithms that involve the process of looking for an item in a container. Searching and sorting go hand in hand. A sorted container will be easier to search. After a container is sorted or ordered, we can apply an apt searching algorithm to find an element. Suppose we need to find the name of the guns which have been used to kill more than 25 enemies. If the container stores the values of the name of the gun and total kills associated with that gun, all we need to do is to first sort that container in ascending number of kills made by the gun. Then we can do a linear search in which we find the first gun which has more 25 kills. Correspondingly the next items in the container after that will have more than 25 kills as the container is sorted. However we can apply better searching techniques.

## Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will find out how we can easily apply searching algorithms to our program.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

Source.cpp

#include <iostream>

#include <conio.h>

using namespace std;

bool Linear\_Search(int list[], int size, int key)

{

// Basic sequential search

bool found = false;

int i;

for (i = 0; i < size; i++)

{

if (key == list[i])

found = true;

break;

}

return found;

}

bool Binary\_Search(int \*list, int size, int key)

{

// Binary search

bool found = false;

int low = 0, high = size - 1;

while (high >= low)

{

int mid = (low + high) / 2;

if (key < list[mid])

high = mid - 1;

else if (key > list[mid])

low = mid + 1;

else

{

found = true;

break;

}

}

return found;

}

## How it works...

Searching for items in a container can happen in many ways. However it matters a lot, if the container has been sorted or not. Let us assume that the container is sorted. The worst way to search an item, is to traverse through the whole container and search for the item. This will take a lot of time for large data sets and absolutely not advisable in game programming. A better way to search for an item is by using binary search. Binary search works by dividing the container in two halves. It checks at the midpoint if the value to be searched is less than or greater than the midpoint value. If it is greater, we can ignore the first half of the container and continue searching only in the second half. Again repeat the process for the second half, by further dividing into two halves. Consequently by doing this, we can reduce the search space of the algorithm immensely. The order of this algorithm is O (log n).

# Finding the complexity of an algorithm

We need an effective way to measure algorithms. That way we will find out whether our algorithm is effective or not. An algorithm will work slower on slower machines and faster on faster machines. Hence computation time is not an effective way to measure algorithms. Algorithms should rather be measured as number of steps. We can call that to be the order of the algorithm. We also need to find out the best case, worst case and average case scenario of the order of the algorithm. This will give us a clearer picture how our algorithm will apply on small sets of data and larger sets of data. Complex algorithms or algorithms of higher order should be avoided as it will increase the number of steps that the device will need to perform the task and hence it slow down the application. Also debugging becomes difficult of such algorithms.

## Getting ready

1. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will find out how easy it is to find the complexity of an algorithm.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#include <iostream>

#include <conio.h>

using namespace std;

void Cubic\_Order()

{

int n = 100;

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

{

for (int j=0; j < n; j++)

{

for (int k = 0; k < n; k++)

{

//Some implementation

}

}

}

}

void Sqaure\_Order()

{

int n = 100;

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

{

for (int j = 0; j < n; j++)

{

//Some implementation

}

}

}

int main()

{

Cubic\_Order();

Sqaure\_Order();

return 0;

}

## How it works...

In this example, we can see how the order of an algorithm or the Big O notation varies with implementation. If we take the first function, Cubic\_Order, the inner most implementation, will take n\*n\*n steps to find the answer. So it has an order of n-cubed O(n^3). This is really bad. Imagine if n is a really large data set, for example let’s say n =1000, it will take 1,000,000,000 steps to find the solution. Avoid cubic order algorithms whenever you can. The second function square\_order, has a square order. The inner most implementation will take n\*n steps to find a solution, so the order of that algorithm is O(n^2). This is again a bad practise.

We should attempt to achieve at least O (log N) complexity. We can achieve log N complexity if we continuously decrease the search space by half, for example Binary Search. There are order algorithms which achieve O (log log N) which is much optimised.

As a general rule, all algorithms following Divide and Conquer will have O (log N) complexity.

# Finding the endian-ness of a device

Endian-ness of a platform refers to the way the most significant byte is stored on that device. This information is highly important as many algorithms can be optimized based on this information. Notably the two most popular rendering SDK, DirectX and OpenGL differ in their endian-ness. The two different types of endian-ness are called big endian and little endian.

## Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

In this recipe, we will find out how easy it is to find the endian-ness of a device.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

Source.cpp

#include <stdio.h>

#include <iostream>

#include <conio.h>

using namespace std;

bool isBigEndian()

{

unsigned int i = 1;

char \*c = (char\*)&i;

if (\*c)

return false;

else

return true;

}

int main()

{

if (isBigEndian())

{

cout << "This is a Big Endian machine" << endl;

}

else

{

cout << "This is a Little Endian machine" << endl;

}

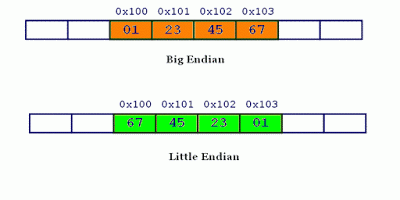
\_getch();

return 0;

}

## How it works...

Little endian is a method of packaging multibyte data types on a machine. In little endian the least significant byte is stored at the start. The other way to package is called big endian. In this method, the least significant byte is stored at the end. In the above example, we are trying to find out whether a machine is little endian or big endian, Based on that we can make appropriate changes and optimise the code. We have used a character pointer called c. We then make c point to an integer i. Since the size of character is 1 byte, when the pointer is deferenced, it will contain only the first byte of the integer. The binary representation of 1 is 01. So if the machine is little endian, the result will be 1 as the last byte is stored first. On the big endian machines, it will give a result of 0, as the first byte or the most significant byte is stored first.  
Suppose integer is stored as 4 bytes, then a variable x with value 0x01234567 will be stored as following:



Insert Image B04929\_04\_01.png

Most of the times compiler takes care of endian-ness, however, endian-ness becomes an issue in network programming if we are sending data from a little endian machine to a big endian machine. Also it becomes an issue if we switch our rendering pipeline from DirectX to OpenGL.

# Using dynamic programming to break down a complex problem

Dynamic programming is a very modern way to solve problems. The process involves breaking a big problem into smaller chunks of overlapping sub-problems, finding solutions for those chunks, store them and all the result of all sub problems to solve the entire complex problem. The sub problems are not independent and have to be combined to form the solution. It is a bit difficult to grasp this technique at first, but will sufficient practice any problem can be solved using dynamic programming. Most of the problems we will encounter while programming a video game, will be complex. Hence mastering this technique will be really useful.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

In this recipe we will find out how easy it is to use dynamic programming to solve a problem

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

Source.cpp

#include<iostream>

#include <conio.h>

using namespace std;

// This is a ternary operation to check which of the expression is higher of the two

int maximum(int x, int y) { return (x > y) ? x : y; }

int AIInventorySack(int TotalWeight, int individual\_weight[], int individual\_value[], int size)

{

if (size == 0 || TotalWeight == 0)

return 0;

else if (individual\_weight[size - 1] > TotalWeight)

return AIInventorySack(TotalWeight, individual\_weight, individual\_value, size - 1);

else return maximum(individual\_value[size - 1] + AIInventorySack(TotalWeight - individual\_weight[size - 1], individual\_weight, individual\_value, size - 1),

AIInventorySack(TotalWeight, individual\_weight, individual\_value, size - 1)

);

}

int main()

{

int individual\_value[] = { 60, 100, 120 };

int individual\_weight[] = { 10, 25, 40 };

int TotalWeight = 60;

int size = sizeof(individual\_value) / sizeof(individual\_weight[0]);

cout << "Total value of sack " << AIInventorySack(TotalWeight, individual\_weight, individual\_value, size);

\_getch();

return 0;

}

## How it works...

This is an example of the classical Knapsack problem. This can be applied in many scenarios in game programming, especially for AI resource management. Let us consider that the total weight (sack) that the AI can carry is a constant. In our example, this is the total weight of the knapsack. Every item that the AI collects in the game has a weight and a value. The AI now needs to decide how to fill up his inventory/sack so that he can sell the total sack for maximum value and get coins.

We solve the problem by recursion by solving for every small combination of items (weight and value) and checking for the maximum value of the two combinations and repeating the process till the total weight of the knapsack is reached.

# Using greedy algorithms to solve problems

Greedy algorithm works by finding the most optimal solution at every stage. So before processing the next step, it will decide its next step based on the previous outcome and the current need of the application. In this way it is better than dynamic programming. However we cannot apply this principle to all problems. Hence greedy algorithm cannot be used for all situations.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

1. In this recipe we will find out how easy it is to use greedy algorithm to solve a problem
2. Open Visual Studio.
   1. Create a new C++ project
   2. Select a win32 console application
   3. Add the following files: Source.cpp
   4. Add the following lines of code.

Source.cpp

#include <iostream>

#include <conio.h>

using namespace std;

void printMaxActivities(int start\_Time[], int finish\_Time[], int n)

{

int i, j;

i = 0;

cout << i;

for (j = 1; j < n; j++)

{

if (start\_Time[j] >= finish\_Time[i])

{

cout << j;

i = j;

}

}

}

int main()

{

int start\_Time[] = { 0, 2, 4, 7, 8, 11 };

int finish\_Time[] = { 2, 4, 6, 8, 9, 15 };

int n = sizeof(start\_Time) / sizeof(start\_Time[0]);

printMaxActivities(start\_Time, finish\_Time, n);

\_getch();

return 0;

}

## How it works...

In this example, we have a set of start time and finish time for different activities. We need to find out which activities can be performed by a single person. We can assume that the container is already sorted based on the finish time. So at every pass, we check whether the current start time is greater than or equal to the previous finish time. Only then can we take up the task. We traverse through the container and keep checking the same condition. Because we are checking at every step, this algorithm is pretty optimised.

# Using Divide and Conquer algorithms to solve problem

In general, divide and conquer is based on the following idea. The whole problem we want to solve may be too big to understand or solve at once. We break it up into smaller problems, find the solution once and then repeat the process. The sub-solutions are independent of each other. Also unlike dynamic programming, we do not store the sub solutions.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. In this recipe we will find out how easy it is to use greedy algorithm to solve a problem
2. Open Visual Studio.
3. Create a new C++ project
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

void Merge\_TwoSets(int \*, int, int, int);

void MergeSortAlgo(int \*a, int low, int high)

{

int mid;

if (low < high)

{

mid = (low + high) / 2;

MergeSortAlgo(a, low, mid);

MergeSortAlgo(a, mid + 1, high);

Merge\_TwoSets(a, low, high, mid);

}

return;

}

void Merge\_TwoSets(int \*a, int low, int high, int mid)

{

int i, j, k, c[100];

i = low;

k = low;

j = mid + 1;

while (i <= mid && j <= high)

{

if (a[i] < a[j])

{

c[k] = a[i];

k++;

i++;

}

else

{

c[k] = a[j];

k++;

j++;

}

}

while (i <= mid)

{

c[k] = a[i];

k++;

i++;

}

while (j <= high)

{

c[k] = a[j];

k++;

j++;

}

for (i = low; i < k; i++)

{

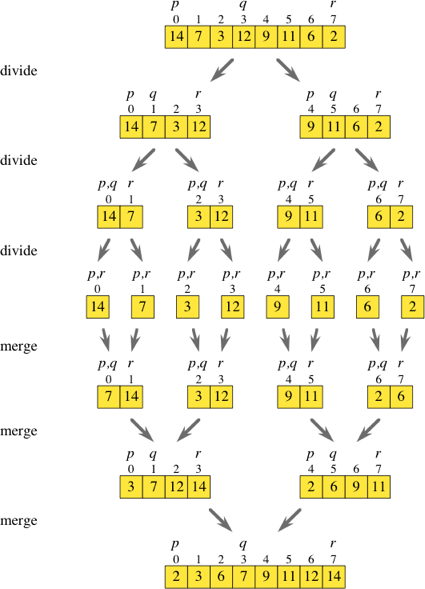
a[i] = c[i];

}

}

How it works...In the above example we have implemented merge sort using the Divide and Conquer technique. Merge sort is very effective and can be used across various problems.

The whole process of Divide and Conquer as we know is dividing the space into smaller solutions. In merge sort, we keep dividing the search space into two parts, individually sort the two parts and merge the solution till we get the final solution. As we can see in the diagram below, the sorting technique involves dividing into smaller solutions.



Insert Image B04929\_04\_02.png

The whole idea behind merge sort is to exploit the fact, that merging two pre-sorted lists is "cheap". Hence this can be used in games and other applications as well.