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.

**Sortinh.h**

1. // Bubble Sort
2. template <class T>
3. void bubble\_sort(T a[], int n)
4. {
5. T temp;
6. for (int i = 0; i<n; i++)
7. {
8. for (int j = 0; j<n - i - 1; j++)
9. {
10. if (a[j]>a[j + 1])
11. {
12. temp = a[j];
13. a[j] = a[j + 1];
14. a[j + 1] = temp;
15. }
16. }
17. }
18. }
19. //Insertion Sort
20. template <class T>
21. void insertion\_sort(T a[], int n)
22. {
23. T key;
24. for (int i = 1; i<n; i++)
25. {
26. key = a[i];
27. int j = i - 1;
28. while (j >= 0 && a[j]>key)
29. {
30. a[j + 1] = a[j];
31. j = j - 1;
32. }
33. a[j + 1] = key;
34. }
35. }
36. //Selection Sort
37. template <class T>
38. int minimum\_element(T a, int low, int up)
39. {
40. int min = low;
41. while (low<up)
42. {
43. if (a[low]<a[min])
44. min = low;
45. low++;
46. }
47. return min;
48. }
49. template <class T>
50. void selection\_sort(T a[], int n)
51. {
52. int i = 0;
53. int loc = 0;
54. T temp;
55. for (i = 0; i<n; i++)
56. {
57. loc = minimum\_element(a, i, n);
58. temp = a[loc];
59. a[loc] = a[i];
60. a[i] = temp;
61. }
62. }
63. //Quick Sort
64. template <class T>
65. int partition(T a[], int p, int r)
66. {
67. T x;
68. int i;
69. x = a[r];
70. i = p - 1;
71. for (int j = p; j <= r - 1; j++)
72. {
73. if (a[j] <= x)
74. {
75. i = i + 1;
76. swap(a[i], a[j]);
77. }
78. }
79. swap(a[i + 1], a[r]);
80. return i + 1;
81. }
82. template <class T>
83. void quick\_sort(T a[], int p, int r)
84. {
85. int q;
86. if (p<r)
87. {
88. q = partition(a, p, r);
89. quick\_sort(a, p, q - 1);
90. quick\_sort(a, q + 1, r);
91. }
92. }

## How it works...

In this example four sorting techniques have been discussed. The four techniques are Bubble sort, Selection sort, Insertion sort and Quick Sort.

Bubble sort is a sorting algorithm that works by continuously traversing through the container to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The process is continued until no more swaps are required. The average, best and worst case scenarios have the order of O (n^2).

Insertion sort is a simple sorting algorithm, a comparison sort in which the sorted container) is built one entry at a time. It is a very simple algorithm to implement. However it is not so effective on large sets of data. The worst and average case scenarios have an order of O (n^2) and the best case scenario, i.e. when the container is sorted, has an order of O (n).

Selection sort is an algorithm which attempts to place an item in its correct position in the sorted list at every pass. The best, worst and average case scenario has an order of O (n^2).

Quick sort is an algorithm which creates a pivot and then sorts the container based on the pivot. Then the pivot is shifted and the process continues. Quick sort is a very effective algorithm and works on almost all real world data and most modern architectures. It makes excellent use of memory hierarchy. Even the inbuilt standard template library uses a modified version of quick sort for its sorting algorithm. The best and average case scenarios for this algorithm is O (n\*log n) and the worst case is O (n^2).

# 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**

1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. bool Linear\_Search(int list[], int size, int key)
5. {
6. // Basic sequential search
7. bool found = false;
8. int i;
9. for (i = 0; i < size; i++)
10. {
11. if (key == list[i])
12. found = true;
13. break;
14. }
15. return found;
16. }
17. bool Binary\_Search(int \*list, int size, int key)
18. {
19. // Binary search
20. bool found = false;
21. int low = 0, high = size - 1;
22. while (high >= low)
23. {
24. int mid = (low + high) / 2;
25. if (key < list[mid])
26. high = mid - 1;
27. else if (key > list[mid])
28. low = mid + 1;
29. else
30. {
31. found = true;
32. break;
33. }
34. }
35. return found;
36. }

## 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.

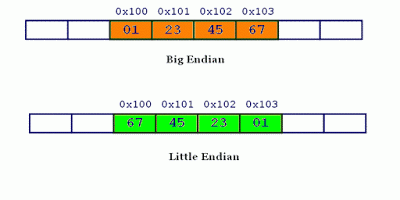
1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

1. #include <stdio.h>
2. #include <iostream>
3. #include <conio.h>
4. using namespace std;
5. bool isBigEndian()
6. {
7. unsigned int i = 1;
8. char \*c = (char\*)&i;
9. if (\*c)
10. return false;
11. else
12. return true;
13. }
14. int main()
15. {
16. if (isBigEndian())
17. {
18. cout << "This is a Big Endian machine" << endl;
19. }
20. else
21. {
22. cout << "This is a Little Endian machine" << endl;
23. }
24. \_getch();
25. return 0;
26. }

## How it works...

Little and big endian are two different ways of how multibyte data types are stored on different machines. In little endian machines, the least significant byte of the multibyte data-type is stored first. On the other hand, in big endian machines, the most significant byte of binary representation of the multibyte data-type is stored first.  
In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced it will contain only first byte of integer. If machine is little endian then \*c will be 1 (because last byte is stored first) and if machine is big endian then \*c will be 0.  
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 problems, finding solutions for those chunks and repeat the process to solve the entire complex problem. 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

## Open Visual Studio.

## Create a new C++ project

1. Select a win32 console application

## Add a source file called Source.cpp

## Add the following lines of code.

Source.cpp

1. #include<iostream>
2. #include <conio.h>
3. using namespace std;
4. int max(int a, int b) { return (a > b) ? a : b; }
5. int knapSack(int TotalWeight, int individual\_weight[], int individual\_value[], int size)
6. {
8. if (size == 0 || TotalWeight == 0)
9. return 0;
10. if (individual\_weight[size - 1] > TotalWeight)
11. return knapSack(TotalWeight, individual\_weight, individual\_value, size - 1);
12. else return max(individual\_value[size - 1] + knapSack(TotalWeight - individual\_weight[size - 1], individual\_weight, individual\_value, size - 1),
13. knapSack(TotalWeight, individual\_weight, individual\_value, size - 1)
14. );
15. }
16. int main()
17. {
18. int individual\_value[] = { 60, 100, 120 };
19. int individual\_weight[] = { 10, 25, 40 };
20. int TotalWeight = 60;
21. int size = sizeof(individual\_value) / sizeof(individual\_weight[0]);
22. cout << "Total value of sack "<<knapSack(TotalWeight, individual\_weight, individual\_value, size);
23. \_getch();
24. return 0;
25. }

## 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.

# Use 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 pieces, solve the pieces separately, and combine the separate pieces together.

## 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

## Open Visual Studio.

## Create a new C++ project

## Add a source file called Source.cpp

## Add the following lines of code.

1. **Source.cpp**
2. #include <iostream>
3. #include <conio.h>
4. using namespace std;
5. const int MAX = 10;
6. class array
7. {
8. private:
9. int arr[MAX];
10. int count;
11. public:
12. array();
13. void add(int num);
14. void makeheap(int);
15. void heapsort();
16. void display();
17. };
18. array ::array()
19. {
20. count = 0;
21. for (int i = 0; i < MAX; i++)
22. arr[MAX] = 0;
23. }
24. void array ::add(int num)
25. {
26. if (count < MAX)
27. {
28. arr[count] = num;
29. count++;
30. }
31. else
32. cout << "\nArray is full" << endl;
33. }
34. void array ::makeheap(int c)
35. {
36. for (int i = 1; i < c; i++)
37. {
38. int val = arr[i];
39. int s = i;
40. int f = (s - 1) / 2;
41. while (s > 0 && arr[f] < val)
42. {
43. arr[s] = arr[f];
44. s = f;
45. f = (s - 1) / 2;
46. }
47. arr[s] = val;
48. }
49. }
50. void array ::heapsort()
51. {
52. for (int i = count - 1; i > 0; i--)
53. {
54. int ivalue = arr[i];
55. arr[i] = arr[0];
56. arr[0] = ivalue;
57. makeheap(i);
58. }
59. }
60. void array ::display()
61. {
62. for (int i = 0; i < count; i++)
63. cout << arr[i] << "\t";
64. cout << endl;
65. }
66. void main()
67. {
68. array a;
69. a.add(11);
70. a.add(2);
71. a.add(9);
72. a.add(13);
73. a.add(57);
74. a.add(25);
75. a.add(17);
76. a.add(1);
77. a.add(90);
78. a.add(3);
79. a.makeheap(10);
80. cout << "\nHeap Sort.\n";
81. cout << "\nBefore Sorting:\n";
82. a.display();
83. a.heapsort();
84. cout << "\nAfter Sorting:\n";
85. a.display();
86. \_getch();
87. }

## How it works...

A heap sorting algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. The heap itself has, by definition, the largest value at the top of the tree, so the heap sort algorithm must also reverse the order. It does this with the following steps:

1. Remove the topmost item (the largest) and replace it with the rightmost leaf. The topmost item is stored in an array.

2. Re-establish the heap.

 3. Repeat steps 1 and 2 until there are no more items left in the heap. The sorted elements are now stored in an array.