**1. Solve the Towers of Hanoi game for the following graph G = (V,E) with V = {Start, Dest, Aux1, Aux2, Aux3} and E = {(Start, Aux1), (Aux1, Dest), (Dest, Aux2), (Aux2, Aux3), (Aux3, Start)}.  
a) Design an algorithm and determine the time and space complexities of moving n disks from Start to Dest.  
b) Implement this algorithm whereby your program prints out each of the moves of every disk. Show the output for n = 1, 2, 3, 4, 5, and 6.**

Tower of Hanoi

**2. Determine for the following code how many pages are transferred between disk and main memory, and assuming each page has 512 words, the active memory set size is 128 (i.e., at any time no more than 128 pages may be in main memory), and the replacement strategy is LRU (the Least Recently Used page is always replaced); also assume that all two-dimensional arrays are of size (1:2048, 1:2048), with each array element occupying one word,  
 for I := 1 to 2048 do  
 {  
 for J := 1 to 2048 do  
 {  
 A[I, J] := A[I,J]\*B[I,J]  
 B[I,J] := C[N-I+1, J]\*A[I,J];  
 }  
 }  
provided the arrays are mapped into the main memory space  
a) in row-major order,  
b) in column-major order.**  
  
  
Row & Col Major

**3. Consider QuickSort on the array A[1:n] and assume that the pivot element x (used to split the array A[lo:hi] into two portions such that all elements in the left portion A[lo:m] are ≤ x and all elements in the right portion A[m:hi] are ≥ x) is the penultimate element of the array to be split (i.e., A[hi-1]).  
Construct an infinite sequence of numbers for n and construct an assigment of the numbers 1..n to the n array elements that causes QuickSort, with the stated choice of pivot, to:  
a) execute optimally (that is A[lo:m] and A[m:hi] are always of equal size),  
b) execute in the slowest possible way.**  
  
  
Quick Sort algorithm

**4. Memory fragmentation in C: Design a C-program that allocates memory for a sequence of 3m arrays of size 100,000 elements each. Then deallocate all odd-numbered arrays and allocate a sequence of m arrays of size 149,000 elements each. Measure the amount of time your program requires for the allocation of the first sequence and for the second sequence. Choose m so that you exhaust almost all of the main memory available to your program. Explain!!**  
  
Memory Allocation using C

**5. Implement a binary search function in three substantially different programming languages. In each program (identical, expect for the programming language), carry out the same 100,000 unsuccessful searches for seven different-sized arrays, namely arrays of sizes 128, 512, 2048, 8192, 32768, 131072, and 524288. Measure in each of the three programs the time it takes to do the 100,000 searches for each of the seven arrays. Compare these timings to the theoretical timings the algorithms binary search provides. Are there differences between the three programs? Explain!!**

|  |  |
| --- | --- |
| **System Specifications** | |
| Operating System | Windows 7 Professional 64-bit |
| Processor | Core 2 Duo T9550 (Mobile CPU, 2.66Ghz, 6MB L2 Cache, 1066Mhz FSB) |
| System Memory | 4GB of DDR2 800Mhz |
| Hard Drive | 320GB 7200RPM SATA 2.5" |

|  |  |  |  |
| --- | --- | --- | --- |
| Language | C++ | Java | Python (64-bit) |
| Repetitions | 1 Billion | 10 million | 10 million |
| Console  Output | Running...  -------------------------  Array Size: 128  Run time: 1.5145 sec  -------------------------  -------------------------  Array Size: 512  Run time: 1.48111 sec  -------------------------  -------------------------  Array Size: 2048  Run time: 1.47868 sec  -------------------------  -------------------------  Array Size: 8192  Run time: 1.47771 sec  -------------------------  -------------------------  Array Size: 32768  Run time: 1.47879 sec  -------------------------  -------------------------  Array Size: 131072  Run time: 1.48409 sec  -------------------------  -------------------------  Array Size: 524288  Run time: 1.47596 sec  -------------------------  Done | Running...  -------------------------  Array Size: 128  Run time: 0.773 sec  -------------------------  -------------------------  Array Size: 512  Run time: 0.954 sec  -------------------------  -------------------------  Array Size: 2048  Run time: 1.143 sec  -------------------------  -------------------------  Array Size: 8192  Run time: 1.326 sec  -------------------------  -------------------------  Array Size: 32768  Run time: 1.53 sec  -------------------------  -------------------------  Array Size: 131072  Run time: 1.738 sec  -------------------------  -------------------------  Array Size: 524288  Run time: 1.939 sec  -------------------------  Done | Running...  -------------------------  Array Size: 128  Run time: 18.1431931 sec  -------------------------  -------------------------  Array Size: 512  Run time: 21.144463842 sec  -------------------------  -------------------------  Array Size: 2048  Run time: 26.1012986513 sec  -------------------------  -------------------------  Array Size: 8192  Run time: 28.7053664884 sec  -------------------------  -------------------------  Array Size: 32768  Run time: 31.8850838917 sec  -------------------------  -------------------------  Array Size: 131072  Run time: 35.6463295345 sec  -------------------------  -------------------------  Array Size: 524288  Run time: 38.6254056 sec  -------------------------  Done |

Observations:

C++:  
C++ runs extremely fast. During testing, arrays ranging from 1 up until the application memory runs out using arrays of type "long", the application consistently runs around 1.5 seconds using 1 billion repetitions. Lowering the repetitions to less than 10 times will result in a seemingly instantaneous runtime for the entire application. This would indicate that the search runs so quick that the O(log n) time complexity to compute the arrays is truncated, and instead what we see is the O(n) time complexity of the binary search iterator. This speed must be due to the lack of an interpreter, the compiler's good use of the CPU architecture, and the CPU architecture itself. Using debug mode on Visual C++ makes the binary search much slower, making it resemble Java's runtime while being a bit slower.

Java:  
Java is much slower than C++ (without debug mode). This is expected, since it uses an interpreter. We can observe a steady increase as the array gets larger. Further down we'll examine if this shows a O(log n) time complexity.

Python:  
Python is the slowest of the three, and It's an interpreted language like Java. The patterns of these results will be examined further below.

Binary.h:

#ifndef BINARY\_H

#define BINARY\_H

#include <iostream>

#include <conio.h>

#include <string>

#include "CTimer.h"

using namespace std;

class Binary

{

private:

CTimer qTimer;

public:

Binary();

void fill(int\*, int);

int binary(int\*, int, int);

};

#endif // BINARY\_H

Binary.cpp:

#include "Binary.h"

int main ()

{

cout << "Running..." << "\n";

Binary();

cout << "Done" << "\n";

\_getch();

return 0;

}

Binary::Binary()

{

int\* A = new int[7]; // Array sizes to be tested

int\* B = NULL; // Test array to be used

A[0] = 128;

A[1] = 512;

A[2] = 2048;

A[3] = 8192;

A[4] = 32768;

A[5] = 131072;

A[6] = 524288;

Binary.cpp (continued):

for (int i=0; i<7; i++)

{

// Fill test array.

B = new int[A[i]];

fill(B, A[i]);

// Start timer

qTimer.Start();

// Perform 1 billion unsuccessful binary searches.

for (long long j=0; j<1000000000; j++)

binary(B, A[i], 0); // Value 0 is never found

// Stop timer

qTimer.End();

cout << "-------------------------" << "\n";

cout << "Array Size: " << A[i] << "\n";

cout << "Run time: " << qTimer.GetTimeInSeconds() << " sec" << "\n";

cout << "-------------------------" << "\n";

// Free memory for current array B

delete [] B;

B = NULL;

}

}

// Fill() fills an array with ascending numbers from 1 to N

// B is the array passed by reference

// N is the size of the ordered array

void Binary::fill(int\* B, int N)

{

int c = 1;

for (int x=0;x<N;x++)

{

B[x] = c;

c++;

}

}

// Binary() performs a single comparison iterative binary search

// B is the array passed by reference

// N is the size of the ordered array

// x is the value to be searched in the array

int Binary::binary(int\* B, int N, int x)

{

int low = 0;

int mid = 0;

int high = N;

while (low < high)

{

mid = low + ((high-low)/2);

if (B[mid] < x)

low = mid + 1;

else

//can't be high = mid-1: here A[mid] >= value,

//so high can't be < mid if A[mid] == value

high = mid;

}

// high == low, using high or low depends on taste

if ((low < N) && (B[low] == x))

return low; // found

else

return -1; // not found

}

Main.java

**public** **class** Main

{

**public** **static** **void** main(String[] args)

{

System.*out*.println("Running...");

Main m = **new** Main();

**int**[] A = {128, 512, 2048, 8192, 32768, 131072, 524288};

**for** (**int** i=0; i<A.length; i++)

{

// Initialize and fill test array.

**int**[] B = **new** **int**[A[i]];

m.fill(B, A[i]);

// Start Timer

**long** startTime = System.*currentTimeMillis*();

// Perform 10 million unsuccessful binary searches.

**for** (**int** j=0; j<10000000; j++)

m.binary(B, A[i], 0); // 0 results in unsuccessful searches

// Stop Timer

**long** stopTime = System.*currentTimeMillis*();

System.*out*.println("-------------------------");

System.*out*.println("Array Size: " + A[i]);

System.*out*.println("Run time: " + ((**double**)(stopTime - startTime))/1000 + " sec");

System.*out*.println("-------------------------");

}

System.*out*.println("Done");

}

// Fill() fills an array with ascending numbers from 1 to N

// B is the array passed by reference

// N is the size of the ordered array

**public** **void** fill(**int** [] B, **int** N)

{

**int** c = 1;

**for** (**int** x=0;x<N;x++)

{

B[x] = c;

c++;

}

}

// Binary() performs a single comparison iterative binary search

// B is the array passed by reference

// N is the size of the ordered array

// x is the value to be searched in the array

**public** **int** binary(**int** [] B, **int** N, **int** x)

{

**int** low = 0; // Lower bound

**int** mid = 0; // Mid point

**int** high = N; //Higher bound

**while** (low < high)

{

mid = low + ((high-low)/2);

**if** (B[mid] < x)

low = mid + 1;

**else**

//can't be high = mid-1: here A[mid] >= value,

//so high can't be < mid if A[mid] == value

high = mid;

}

// high == low, using high or low depends on taste

**if** ((low < N) && (B[low] == x))

**return** low; // found

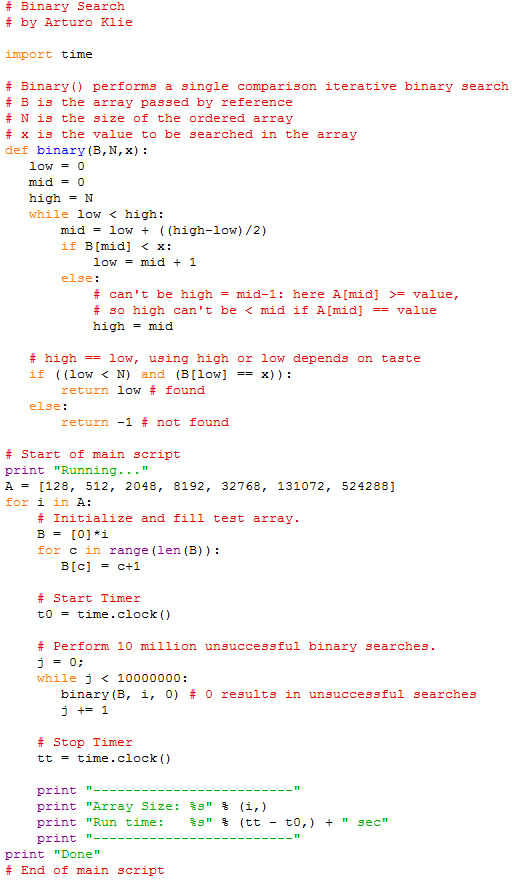
**else**

**return** -1; // not found

}

}

p5.py:



Here is the windows timer implementation used for the C++ programs:

Ctimer.h:

#include <iostream>

#include <windows.h>

class CTimer

{

public:

CTimer() {

QueryPerformanceFrequency(&mqFreq);

}

~CTimer() {}

void Start() {

QueryPerformanceCounter(&mqStart);

}

void End() {

QueryPerformanceCounter(&mqEnd);

}

double GetTimeInSeconds() {

return (mqEnd.QuadPart - mqStart.QuadPart)/(double)mqFreq.QuadPart;

}

double GetTimeInMilliseconds() {

return (1.0e3\*(mqEnd.QuadPart - mqStart.QuadPart))/mqFreq.QuadPart;

}

double GetTimeInMicroseconds() {

return (1.0e6\*(mqEnd.QuadPart - mqStart.QuadPart))/mqFreq.QuadPart;

}

double GetTimeInNanoseconds() {

return (1.0e9\*(mqEnd.QuadPart - mqStart.QuadPart))/mqFreq.QuadPart;

}

private:

LARGE\_INTEGER mqStart;

LARGE\_INTEGER mqEnd;

LARGE\_INTEGER mqFreq;

};