Pre-Technical Assessment Test

Problem Statement - 1

Compare Cheney's algorithm and Mark-compact algorithm with C++ and demonstrate which
is better and why?
A C++ Code demonstrating the principles of both this algorithm is expected. Documentation with comparison of both these algorithms including the Code, Graphs, and explanation is expected.

Garbage collection:

Cleans up object and memory space when you're done with it. No more free cells.

Form of Garbage collection:

Mark-compact algorithm:

A mark-compact algorithm is a type of garbage collection algorithm used to reclaim unreachable memory. Mark-compact algorithm is a combination of the mark-sweep algorithm and Cheney's copying algorithm.

Steps:

1. Recursively mark all arrays reachable from the roots as visited.
2. For each visited array, compute a new address for the array in the compacted heap.
3. Update pointers to point to the new addresses.
4. Relocate each array to its new address.

Algorithm:

The Mark-Compact algorithm is composed of 4 sub-routines.

markcompact()

```
foreach r in roots
        mark(r)
computeAddresses()
updatePointers()
relocate()
```

Recursively mark all arrays that are reachable from the array beginning at address addr.

markaddr()

```
let header = heap[addr]
if visited(header) == false then
        visited(header) ← true
        n ← size(header)
        foreach i in 1 .. n
                child ← heap[addr+i]
                if pointer?(child) then mark(child)
```

For each visited array, compute a new address for the array in the compacted heap.

computeAddress()

```
scan ← 1
new ← 1
while scan <= #heap
        let header = heap[scan]
        n ← size(header)
        if visited(header) == true then
                forwardingAddress(header) ← new
        new ← new + 1 + n
scan ← scan + 1 + n
```

Cheney's algorithm:

Cheney, is a stop and copy method of tracing garbage collection in computer software systems. In this scheme, the heap is divided into two equal halves, only one of which is in use at any one time.

Steps:

1. Memory space is divided in two:
   From-Space and To-Space.
2. New arrays are allocated in From-Space.
3. When From-Space becomes full, program execution stops, and all live data is copied to To-Space.
4. From-Space and To-Space are flipped and program execution continues.

Algorithm:

Copy the array beginning at location from in from-space to the next free locations in to-space

```
let header = fromSpace[from]
n ← size(header)
foreach i in 0..n
    toSpace[free] ← fromSpace[from+i]
    free ← free + 1
```

If the array has not been visited, copy it to to-space and set the forwarding address. Return the tospace location of the array.

```
let header = fromSpace[from]
if visited(header)
    return forwardingAddress(header)
else
    addr ← free
    copyArray(from)
```

```
        visited(header) ← true
        forwardingAddress(header) ← addr
        return addr

scan ← 1
free ← 1
foreach r in roots
        r ← copy(r)
while scan < free
        let x = toSpace[scan]
        if pointer?(x)
                toSpace[scan] ← copy(x)
                scan ← scan + 1
        fromSpace, toSpace ← toSpace, fromSpace
```

Problem Statement - 2

Do zipping operation on folder which contains multiple files : Write a C++ code to zip the folder with files, from scratch or using C++ library

```
void CreateEmptyZipFile(CString strPath)
{
	BYTE startBuffer[] = {80, 75, 5, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0};
	FILE *f = _wfopen(strPath.GetBuffer(strPath.GetLength()),
_T("wb"));
	strPath.ReleaseBuffer();
	fwrite(startBuffer,sizeof(startBuffer),1,f);
	fclose(f);
}

void ZipFile(CString strSrc, CString strDest)
{
	//Create an empty zip file
	CreateEmptyZipFile(strDest);

	BSTR bstrSource = strSrc.AllocSysString();
	BSTR bstrDest = strDest.AllocSysString();

	HRESULT hResult = S_FALSE;
	IShellDispatch *pIShellDispatch = NULL;
	Folder *pToFolder = NULL;
	VARIANT variantDir, variantFile, variantOpt;

	CoInitialize(NULL);
```

```
	hResult = CoCreateInstance(CLSID_Shell, NULL,
CLSCTX_INPROC_SERVER,
		IID_IShellDispatch, (void **)&pIShellDispatch);

	if (SUCCEEDED(hResult))
	{
		VariantInit(&variantDir);
		variantDir.vt = VT_BSTR;
		variantDir.bstrVal = bstrDest;
		hResult = pIShellDispatch->NameSpace(variantDir,
&pToFolder);

		if (SUCCEEDED(hResult))
		{
			VariantInit(&variantFile);
			variantFile.vt = VT_BSTR;
			variantFile.bstrVal = bstrSource;

			VariantInit(&variantOpt);
			variantOpt.vt = VT_I4;
			variantOpt.lVal = FOF_NO_UI;

			hResult = pToFolder->CopyHere(variantFile,
variantOpt);
			Sleep(1000);
			pToFolder->Release();
		}

		pIShellDispatch->Release();
	}

	CoUninitialize();
```

```
}
```

UNZIP File

```
void UnZipFile(CString strSrc, CString strDest)
{
	BSTR source = strSrc.AllocSysString();
	BSTR dest = strDest.AllocSysString();

	HRESULT hResult = S_FALSE;
	IShellDispatch *pIShellDispatch = NULL;
	Folder *pToFolder = NULL;
	VARIANT variantDir, variantFile, variantOpt;

	CoInitialize(NULL);

	hResult = CoCreateInstance(CLSID_Shell, NULL,
CLSCTX_INPROC_SERVER,
		IID_IShellDispatch, (void **)&pIShellDispatch);
	if (SUCCEEDED(hResult))
	{
		VariantInit(&variantDir);
		variantDir.vt = VT_BSTR;
		variantDir.bstrVal = dest;
		hResult = pIShellDispatch->NameSpace(variantDir,
&pToFolder);

		if  (SUCCEEDED(hResult))
		{
			Folder *pFromFolder = NULL;
			VariantInit(&variantFile);
	  variantFile.vt = VT_BSTR;
	  variantFile.bstrVal = source;
```

```
                pIShellDispatch->NameSpace(variantFile,
&pFromFolder);

                FolderItems *fi = NULL;
                pFromFolder->Items(&fi);

                VariantInit(&variantOpt);
        variantOpt.vt = VT_I4;
        variantOpt.lVal = FOF_NO_UI;

                VARIANT newV;
                VariantInit(&newV);
                newV.vt = VT_DISPATCH;
                newV.pdispVal = fi;
                hResult = pToFolder->CopyHere(newV, variantOpt);
                Sleep(1000);
                pFromFolder->Release();
        pToFolder->Release();
            }
            pIShellDispatch->Release();
    }

    CoUninitialize();

}
```