BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS-F211: Data Structures and Algorithms

Lab 5: Recursion vs Iteration

Introduction

Welcome to week 5 of Data Structures and Algorithms! Today's lab will be focussing on practical issues related to the time performance of algorithms. There are practical constraints which impose bounds on approaches that theoretically should perform better. These will form the subject matter of today's lab. You will be making some use of concepts that you have gained exposure to in the previous labsheets (such as linked lists and stacks) as well as the lectures. Hopefully, this discussion will open your eyes to a new way of inspecting the performance of algorithms.

Topics to be covered in this labsheet

- Recursion vs Iteration
- Tail-Recursive Algorithms
    - Introduction
    - Case studies (to synthesise a general approach for conversion)
    - Making use of explicit stack (a brief introduction)
- Locality Awareness
    - Spatial Locality Aware Algorithms

*<u>Note:</u> All the additional files mentioned in the problems in this labsheet are available in the **input_files** subdirectory.*

Recursion vs Iteration

Recursive algorithms are natural and intuitive for solving several problems (especially problems whose solutions are of the divide-and-conquer kind), but you have seen in the lectures how recursive algorithms incur a stack space overhead and end up taking more space and time than they ought to. This is due to the extensive use of the call stack that ends up happening in recursive algorithms. There are so many more stack operations that need to be performed and so many (possibly even unnecessary) stack frames that need to be delicately maintained (internally) during a recursive function's execution that it ends up taking a lot more time than we would expect (and desire) it to. For more details go through the slides on the lecture(s) related to "Recursion vs Iteration".

As a rule of thumb, therefore, a recursive function is always going to be slower than an iterative function implementing the same approach. *This is because of the overhead of maintaining so many stack frames.* This means that wherever we see a recursive function, there is possibly some scope for performance improvement if only we can write an iterative function to solve the same instead.

Let us understand the same with the help of the following example of **binary search**.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

// Recursive binary search
int binarySearchRecursive(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearchRecursive(arr, l, mid - 1, x);
        return binarySearchRecursive(arr, mid + 1, r, x);
    }
    return -1;
}
```

```c
// Iterative binary search
int binarySearchIterative(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r - l) / 2;
        if (arr[m] == x)
            return m;
        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}

// Driver code
int main(void)
{
    int arr[500000];
    for(int i = 0; i < 500000; i++)
        arr[i] = i*2;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result[5000];

    struct timeval t1, t2;
    double time_taken;
    gettimeofday(&t1, NULL);
    for(int j = 0; j < 5000; j++)
        result[j] = binarySearchIterative(arr, 0, n - 1, j*8);
    gettimeofday(&t2, NULL);
    time_taken = (t2.tv_sec - t1.tv_sec) * 1e6;
    time_taken = (time_taken + (t2.tv_usec - t1.tv_usec)) * 1e-6;

    printf("Iterative binary search took %f seconds to execute\n", time_taken);

    gettimeofday(&t1, NULL);
    for(int j = 0; j < 5000; j++)
        result[j] = binarySearchRecursive(arr, 0, n - 1, j*8);
```

```
    gettimeofday(&t2, NULL);
    time_taken = (t2.tv_sec - t1.tv_sec) * 1e6;
    time_taken = (time_taken + (t2.tv_usec - t1.tv_usec)) * 1e-6;

    printf("Recursive binary search took %f seconds to execute\n", time_taken);

    return 0;
}
```

In this example, we have implemented both recursive as well as iterative versions of binary search. Then, we executed them on a simple array of the first 500,000 even numbers (the number is large for the time difference to be more apparent). Obviously, the array is sorted (it has to be). We performed search for not just one element but 5,000 elements situated near the start of the array (we are doing this for multiple elements to get a more significant time difference). We measure the times taken by both algorithms, and report them on stdout.

Following are a few sample executions of the above code:

```
Iterative binary search took 0.000438 seconds to execute
Recursive binary search took 0.000883 seconds to execute
```

```
Iterative binary search took 0.000444 seconds to execute
Recursive binary search took 0.000751 seconds to execute
```

```
Iterative binary search took 0.000434 seconds to execute
Recursive binary search took 0.000766 seconds to execute
```

You may execute the program yourself as many times as you like to verify the results. But we can clearly see that, on average, the recursive binary search takes up more time than its iterative counterpart for the same task.

Task 1: Write a function that recursively generates the n'th Fibonacci number. Also, write a function that does the same iteratively. Now, first call the recursive function for a sample input of n=10,000 and measure the time taken to generate the number. Then, call the iterative function for the same input and measure the time taken by it. Compare the results to prove the point that the recursive function takes more time than the iterative one. [Note: You may need to alter the input to get desirable results.]

Another example of the same is **insertion sort**.

```c
#include <stdio.h>

// recursive insertion sort
void insertionSortR(int *arr, int n)
{
    if (n <= 1) return;
    insertionSortR(arr, n-1);
    int last = arr[n-1];
    int j = n-2;
    while (j >= 0 && arr[j] > last)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}

// iterative insertion sort
void insertionSortI(int *arr, int n)
{
    for (int i = 1; i < n; i++)
    {
        int last = arr[i];
        int j = i-1;
        while (j >= 0 && arr[j] > last)
        {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = last;
    }
}
```

In the recursive version, we simply have a base case and a recursive case (discerned with an if statement at the start). In the base case, we do nothing (sorting is already done when our input is of size 1). In the recursive case (for some size n > 1), we first sort the first n-1 elements

(through our recursive call), and then place the n'th element within the sorted n-1 subarray, in order, with the help of a while loop. This is our recursive approach.

Whereas, in the iterative version, we use a single for loop to gradually build up a sorted subarray at the start end of the array. As our sorted subarray increases in size, we also have to perform a while loop to iterate through to the last element of the sorted subarray, so that we can perform the insert-in-order operation with the last element.

Task 2: We have provided you with both the recursive and iterative implementations of insertion sort in the code snippet above. Write a suitable main() function to test them and measure the time they take for a sample integer array and again prove the point that the recursive variant ends up taking more time than its iterative counterpart. [You have to use a suitably long input array to be able to see the time difference between the execution of the two functions. We have provided you with input files containing a large number of random integers. You may use them as sample input. We have also provided you with a template program *number_reader.c* which reads one of these files and stores the integers into an array.]

Home Exercise 1: Design the mergeSort() function that implements the merge sorting algorithm on an array (you may simply borrow the code for this from Lab 4). Now, design an iterative version of mergeSort(). For this, you may take help of your lecture notes. Once these two functions have been designed, write a suitable main() function to execute them and measure the time taken by them for sorting an input array. Use the measured times to establish that the recursive function takes more time to execute than the iterative equivalent. [You have to use a suitably long input array to be able to see the time difference between the execution of the two functions. We have provided you with input files containing a large number of random integers. You may use them as sample input. We have also provided you with a template program *number_reader.c* which reads one of these files and stores the integers into an array.]

Tail Recursive Algorithms

Introduction

Often, we write recursive functions in which the recursive call made to the function is followed by an operation that is performed on its return value. Such functions are *non-tail recursive* (NTR) in the sense that the recursive call is *not* at the absolute tail-end of the function.

```
int sum(int n)
{
    if (n == 0)
        return 0;
```

```
    return n + sum(n - 1);
}
```

For example, the above function is *non-tail recursive (NTR)*, because here we are performing an operation (addition) after the recursive call returns. Note how **sum(n-1)** has to be followed by an addition between **n** and whatever value was returned by **sum(n-1)**.

Whereas, if a recursive function does have its recursive call occurring at the tail end of the function with no operation being performed after it, such a recursive function is *tail recursive* (TR).

```
int sum(int n, int k)
{
    if (n == 0)
        return k;
    return sum(n - 1, n + k);
}
```

For example, the above function is *tail recursive (TR)* because there is no further operation that has to be performed after the recursive call to **sum(n-1, n+k)** returns a value. In other words, the return from the recursive call is the last operation to be performed in this function. [The process of conversion from an NTR function to a TR function is an important concept and will be explained very thoroughly in the next section.]

You have already learnt in the lectures that TR functions are easier to convert to an iterative function (make sure that you are clear with the definitions of the types of functions mentioned above before you proceed with this section as we will be referring to them multiple times). You can revise these concepts from the slides on the lecture(s) related to “Recursion vs Iteration”.

As we have seen and proved multiple times in the previous section, iterative functions are more time-efficient than recursive functions for the same problem. This makes TR functions very important for us, because, wherever they occur, we can follow a systematic approach to convert them into iterative functions and consequently do better on time. There are also certain optimisations that the compiler can perform (that you have learnt in the lectures) on a TR function to make it run faster.

However, since most of our recursive functions are NTR, it would also be good if we could convert any NTR function to its TR equivalent. And true, this can be done sometimes, but not all

of the time. *Not* all NTR functions are convertible to their TR equivalent. Typically, if an NTR function is doing the same thing to the value returned by the recursive function call each time, irrespective of what is being returned, then it is quite likely that we can convert it to a TR function. However, if an NTR function does different things to the value returned by the recursive function call depending on the value itself that is being returned, then it becomes quite difficult to convert that to a TR function.

Case Studies

In this section, we shall see several examples of the above to acquaint ourselves with the concepts and the process of transformation.

We should start with a simple example. Consider the following function to **compute the factorial of a number**.

```
// factorial of a number without tail recursion
int factNTR(int n) {
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * factNTR(n - 1);
    }
}
```

Clearly, this is a function that implements recursion to compute the factorial of a number the way we would have written it so far. Its working is intuitively clear to us, and of course, it executes correctly. However, notice that the recursive call that is being made by the function is not the very last thing that is happening. The return value from the recursive call is *multiplied* with something else. Because of this, we conclude that the above function is non-tail recursive.

It would be good if we could convert this function to a tail recursive form, wouldn't it? Let us try to think of a strategy for the same.

Note how in this function, something similar is always being done to the value returned by the recursive function. We are multiplying the value returned from the called function with the input to the caller function. We must leverage this in our conversion from NTR to TR. If we add

another input parameter to the function that will hold the value of the multiplication at each step of the recursion, then we can keep passing it from one call to the next, recursively, and obtain the same result. Let us see how that might look programmatically.

```
// factorial of a number by implementing tail recursion
int factTR(int n, int acc) {
    if (n == 0)
    {
        return acc;
    }
    else
    {
        return factTR(n - 1, n * acc);
    }
}
```

Try to go through the working of the above function step by step and understand exactly how it executes. The most significant idea here is the inclusion of an **accumulator**. This way, we are no longer performing the multiplication *after* the return of the recursive call. Rather, now we are doing it *before* making the recursive call. And we have made this possible by passing the accumulator as a parameter to the function itself. Of course, this means that when calling our function we now have to pass a second parameter to it, but in the above case, it is clear that that has to be 1 (one), because this is a multiplicative accumulator. [Note: It is not at all required, but for convenience, we may create a wrapper function around factTR() that calls factTR() internally (abstraction) by passing the default 1 as the second parameter, so that the client need not worry about this additional second parameter that we have included (this of course adds one extra stack frame to the call stack). That would look like `int fact(int n){return factTR(n,1);}`]

This conversion that we have performed is actually remarkable because now we can convert this recursive function to an iterative function very easily. However, already the tail recursive function might perform much better than the earlier non-tail recursive one (due to optimisations performed by the compiler on it). But we can do better; let us take a look at the end result of the conversion to iterative.

```
// conversion of tail recursive factorial to iterative
int factI(int n)
{
```

```
    int acc = 1;
    while (n > 0)
    {
        acc = acc * n;
        n = n - 1;
    }
    return acc;
}
```

Notice how we have simply unwrapped the factTR() function to obtain this simplistic iterable construct. We saw the repetitive nature of the recursive call in factTR() and therefore saw a way to make this clever conversion. This iterative function would undoubtedly take lesser time to execute than the two functions above it both of which implement recursion.

Task 3: Copy all the above three functions into a .c file and write a suitable driver function to test their working and measure the time taken by them. Once again, compare the times taken to understand the importance of the conversion that we have just made. You can try with very large values of n to observe a significant difference in the execution time.

Another example to illustrate this conversion is one where we want to find the **sum of all the elements in a linked list of integers**.

Consider the following snippet of code (we have included the definitions of linked list and its creation function from the first labsheet).

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node * NODE;
struct node{
    int ele;
    NODE next;
};

typedef struct linked_list * LIST;
struct linked_list{
    int count;
    NODE head;
```

```
};

LIST createNewList()
{
    LIST myList;
    myList = (LIST) malloc(sizeof(struct linked_list));
    myList->count=0;
    myList->head=NULL;
    return myList;
}

NODE createNewNode(int value)
{
    NODE myNode;
    myNode = (NODE) malloc(sizeof(struct node));
    myNode->ele=value;
    myNode->next=NULL;
    return myNode;
}

// Recursive function to compute sum of integers in a linked list
int llSumNTR(NODE head)
{
    if (head == NULL)
        return 0;
    return head->ele + llSumNTR(head->next); // Pay close attention here
}

// This is just a wrapper function to take LIST as input, but run the recursive
sum function on its NODEs starting from the head
int llSumNTRWrapper(LIST list)
{
    return llSumNTR(list->head);
}
```

Clearly, this is non-tail recursive. But following a process similar to the one indicated earlier, we can convert this function to an iterative form as well. This is your next task.

Task 4: Convert the above non-tail recursive function first to a tail recursive function that performs the same task, and then convert the tail recursive function to its iterative equivalent. You may create wrapper functions for these three for your convenience. Measure the times taken by all three and compare them quantitatively. [We have provided you with input files containing a large number of random integers. You may use them as sample input. We have also provided you with a template program *LL_reader.c* which reads one of these files and creates a linked list with them. It has the above snippet of code also pasted onto it. Use that program as a template for the task.]

[Hint: Here, the accumulator will need to hold the sum instead of the product; given this, what should the initial value given to the accumulator parameter be?]

Home exercise 2: Consider the following non-tail recursive function to measure the length of an input string:

```
int lengthNTR(char *str)
{
    if (*str == '\0')
    {
        return 0;
    }
    else
    {
        return 1 + lengthNTR(str + 1);
    }
}
```

Perform a series of transformations (similar to the ones we have described previously) on `lengthNTR()` to convert it first into `lengthTR()`, a tail recursive equivalent of `lengthNTR()`, and then finally `lengthI()`, the iterative equivalent of `lengthTR()`. Then, execute them on sufficiently large sample inputs and measure the time taken by them to execute. **[Note: You are not allowed to use any library function at any stage of the conversion.]**

Note that with enough practice, it becomes easier to convert a NTR function directly to an iterative function, without going through the intermediary TR function, by merely observing. However, to get to that stage we must first familiarise ourselves with the standard procedure that we have discussed above and then get enough practice with the same.

A Brief Introduction to Making Use of Explicit Stack

The problem that makes a recursive function slower is that it demands a stack space overhead and ends up taking more space and time than it ought to. However, now that we have learnt about stacks in week 3, can we not make use of an explicit stack to manually perform our recursive computations as we desire? In this attempt, we will also end up erasing the recursion entirely, therefore making the function iterative. If we simulate our call stack using an *explicit stack*, we can execute any recursive function iteratively in a single main loop. The resulting code, however, will be quite unpleasant to read. This is why the general process of conversion is typically not preferred, and it is most ideal if we can convert our NTR function first to a TR function. That way, the resulting iterative function is still fairly readable (you have already seen this approach in great detail in the prior examples). However, since NTR to TR conversion is not always possible, we must also somewhat familiarise ourselves with the explicit stack method.

The fundamental idea here is that we are creating an explicit stack on our own to simulate whatever happens to the recursive call stack. The general process is:

- Recursive calls get replaced by “push”
    - depending on the details, may push new values, old values, or both
- Returns from recursive calls get replaced by “pop”
- The main calculation of the recursive routine gets put inside a loop
    - at start of the loop, set variables from stack top and pop the stack

A detailed example of this process will be covered in lectures in which you will first learn the Quick sort algorithm and then you will also see how it is converted to an iterative form through the use of an explicit stack. This will also be covered in the next labsheet.

Locality Awareness

Spatial Locality Aware Algorithms

The concept of spatial locality is that data accessed at some point in a program is situated next to data that is likely to be accessed in the near future. This concept becomes very important for us while designing our algorithms (solutions) to problems, because our computer architecture is usually such that data is spatially cached and pre-fetched. This means that while accessing data from a certain location in memory, our computers prefetch the entire block that the requested data belongs to and place the block in the cache. This makes sense because the computer expects that spatial locality will be maintained more often than not (and indeed, it is). Therefore, if later we request data from somewhere next to the data we requested earlier, then the computer need not fetch another entire block from memory, because the block that this adjacent data belongs to has already been *pre-fetched* when we requested for the earlier data. If we do this more often than not, then we are saving time simply due to our memory hierarchy and computer organisation.

These ideas have to be kept in mind while desiging our algorithms. An algorithm which takes into account the concept of spatial locality to leverage the time boost that the memory hierarchy may consequently provide is known as a *spatial locality aware algorithm*. Whereas, an algorithm which does not, is known as a *non spatial locality aware algorithm*. Of course, a spatial locality aware algorithm will have better practical performance than a non spatial locality aware algorithm for the same problem.

It becomes easier to understand the same with the help of examples.

Suppose our problem is to store a fixed number of integers in a sequence on which we will frequently have to perform a **sequential scan** through all the elements. For this problem, two approaches immediately come to mind. We can store the numbers in a typical array, or we can store them in a linked list.

Based on the concept of spatial locality that we have just learnt, it would make more sense to store the numbers in a fixed-length array rather than a linked list of nodes. This is because the nodes in a linked list might very well be stored in wholly different locations in memory. Whereas, a fixed-length array's elements are always stored contiguously. This means that our linked list approach is *non spatial locality aware*. It does not leverage the spatial locality concept. On the other hand, our fixed-length array approach is in fact *spatial locality aware*. It is making use of the spatial locality concept to leverage the memory hierarchy and perform better.

Let us actually implement this and compare the times taken by the two approaches. We already have the linked list prototype from labsheet 1. Take that prototype (including the allied functions), copy it into a file named *seq.c*, and write the following main function into it.

```
int main()
{
    LIST myList = createNewList();
    int n = 100000;
    int arr[n];
    int i;
    for(i=0; i<n; i++)
    {
        arr[i] = i;
    }

    for(i=0; i<n; i++)
    {
        NODE myNode = createNewNode(arr[i]);
        insertAfter(i-1, myNode, myList);
    }

    struct timeval start, end;
    int temp1;
    gettimeofday(&start, NULL);
    for(i=0; i<n; i++)
        temp1 = arr[i];
    gettimeofday(&end, NULL);
    printf("Time taken for sequential scan on array: %ld microseconds\n",
((end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 +
start.tv_usec)));

    gettimeofday(&start, NULL);
    NODE temp = myList->head;
    while(temp != NULL)
        temp = temp->next;
    gettimeofday(&end, NULL);
    printf("Time taken for sequential scan on linked list: %ld microseconds\n",
((end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 +
start.tv_usec)));
}
```

Make sure you have included the required libraries. Also, you may need to alter the value of **n** here to obtain desirable results for your system. At small values of **n**, you would expect both approaches to take less than a microsecond of time on a reasonably fast system. At the same time, there is a limit to how large an **n** we can procure from our operating system due to memory constraints. Also note how this "appropriate" value of **n** will vary from system to system. On our system, we had to experiment a bit with **n** and ended up obtaining decent results for n = 100000. Following are three sample executions of the above program:

```
Time taken for sequential scan on array: 271 microseconds
Time taken for sequential scan on linked list: 1516 microseconds
```

```
Time taken for sequential scan on array: 287 microseconds
Time taken for sequential scan on linked list: 1001 microseconds
```

```
Time taken for sequential scan on array: 266 microseconds
Time taken for sequential scan on linked list: 1206 microseconds
```

Another classic example illustrating the same concept is that of **matrix addition**.

We all know that matrix addition involves two nested for loops to scan through the two dimensions of the 2D arrays (or matrices) and add each corresponding element. We also know that the elements of the 2D array are stored in a row-major format in memory. There can be a difference in the ordering of the nested for loops, as shown in the below pseudocode:

```
// Row-by-row (or row-wise) addition
for (i=0; i<N; i++)
      for(j=0; j<N; j++)
            M3[i,j] = M2[i,j] + M1[i,j]

// Col-by-col (or col-wise) addition
for (j=0; j<N; j++)
      for(i=0; i<N; i++)
            M3[i,j] = M2[i,j] + M1[i,j]
```

Clearly, the second approach should perform worse than the first since it is *non spatial locality aware*. In a row-major arrangement, if we are scanning column-wise (column by column), then it is not going to leverage the spatial locality concept (given that our rows are long enough that the entire matrix does not fit into our cache). On the other hand, given that our setup is in

row-major form, if we decide to scan row-wise (row by row), then we are taking benefit of the spatial locality concept, thereby making our approach *spatial locality aware*.

Task 5: Implement the two different kinds of matrix additions described above. Take a sufficiently large pair of input matrices and add them using both the prescribed methods. Measure the time taken by each method. Compare their performance to establish that the non spatial locality aware approach takes more time for a sufficiently large input. [Refer to the program *matrix_generator.c* which we have provided. It asks the user for 'n' as input and creates a random square matrix of size n×n. Build upon this to implement matrix addition on two randomly generated matrices of same size. You can implement the two different kinds of matrix additions as two different functions and then compare their running times.]

Home Exercise 3: Arrays of strings are a very important requirement for us as students of data structures and algorithms. However, there are two ways of implementing this in C. Either we have a simple fixed 2D array (matrix) of characters, or we have a 1D array of character pointers (strings). Based on what you have learnt in this labsheet, you can make an educated comment about the above two approaches now.
2a: Theoretically discuss which of the two approaches would be a better approach and for what requirement.
2b: Write a program in which you have implemented both approaches and you are comparing the time taken for both for performing lexicographical sorting (attaining dictionary order for the strings) on the strings that they hold and see if your theoretical discussion matches with the empirical result you obtained.

Home Exercise 4: One of the most remarkable examples of the importance of the two concepts that we learnt today (worsened time efficiency for recursive functions and spatial locality awareness to improve efficiency) is that of **Strassen's matrix multiplication**.

It is a divide-and-conquer algorithm for matrix multiplication. The first key idea here is that matrix multiplication can be divided. Refer to figure 1 below to understand the same.

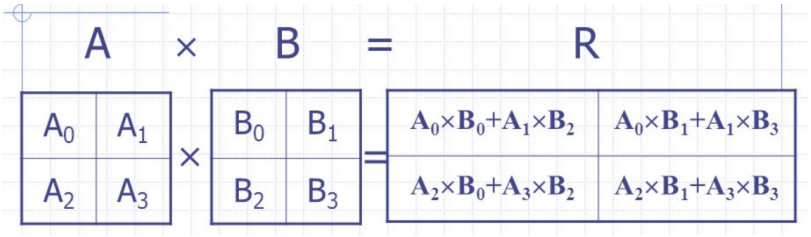

Figure 1: Illustrating the divide-and-conquer approach for matrix multiplication

A, B, and C are square matrices of size N x N.
A_0 to A_3 are submatrices of A and are of size N/2 x N/2 each.
B_0 to B_3 are submatrices of B and are of size N/2 x N/2 each.

The approach is for us to divide our matrices into submatrices, as shown, and to multiply the submatrices for the resultant matrix. Recursively perform this division until you hit the base-case matrix size of 2 x 2 where you will use the blocked multiplication equations given below. In each step, keep constructing the resultant matrix using the equations as shown above.

The blocked matrix multiplication equations are the following for normal matrix multiplication of 2 x 2 matrices (base case):

$$R_{11} = A_{11}\ B_{11} + A_{12}\ B_{21}$$
$$R_{12} = A_{11}\ B_{12} + A_{12}\ B_{22}$$
$$R_{21} = A_{21}\ B_{11} + A_{22}\ B_{21}$$
$$R_{22} = A_{21}\ B_{12} + A_{22}\ B_{22}$$

However, Strassen came up with the following set of blocked matrix multiplication equations for the 2 x 2 case:

$$R_{11} = S_1 + S_4 - S_5 + S_7$$
$$R_{12} = S_3 + S_5$$
$$R_{21} = S_2 + S_4$$
$$R_{22} = S_1 + S_3 - S_2 + S_6$$

where:

$$S_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$$
$$S_2 = (A_{21} + A_{22}) * B_{11}$$
$$S_3 = A_{11} * (B_{12} - B_{22})$$
$$S_4 = A_{22} * (B_{21} - B_{11})$$
$$S_5 = (A_{11} + A_{12}) * B_{22}$$
$$S_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$
$$S_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

There are only *seven* matrix multiplication operations in the method proposed by Strassen, as compared to the normal matrix multiplication which has *eight*. Since multiplication is the dominant operation here, Strassen's method ends up having a slightly better time complexity than the normal matrix multiplication method. The normal matrix multiplication method has a time complexity of $\mathbf{O(n^3)}$, while Strassen's version has a time complexity of $\mathbf{O(n^{\lg 7})}$ which is approximately $\mathbf{O(n^{2.81})}$. This can be verified using Master's Theorem. You may refer to the following link for the same: https://pages.cs.wisc.edu/~jyc/3.MASTER.pdf (the relevant slides

are on pg. 24-37). You can imagine that the difference in runtime becomes significant as 'n' becomes large.

However, when 'n' becomes large, another problem arises. Notice how the divide-and-conquer method is not cache sensitive. It is *not* spatial locality aware. Therefore, *even though* theoretically speaking Strassen's method should outperform the normal matrix multiplication, *practically* it ends up massively underperforming. There is another reason that adds to its underperformance, and that is the fact that it is a recursive algorithm (whereas we typically think of normal matrix multiplication as a triple for loop iterative task). Due to both of the above reasons, Strassen's multiplication method becomes a classic victim of the practical flaws of recursive and locality unaware approaches, thereby justifying all the discussions we have had in this labsheet.

Now, your task is to implement three approaches for matrix multiplication:

(a) Normal iterative matrix multiplication.
(b) Employing the recursive divide and conquer strategy, but for normal matrix multiplication (ie., use the blocked set of equations above, ie., in this case, the one having eight multiplications in total).
(c) Employing the recursive divide and conquer strategy, but with Strassen's matrix multiplication equations (ie., use the blocked set of equations having seven multiplications in total).

For each of the three approaches, take a sample pair of matrices (refer to *matrix_generator.c* for creating the sample matrices) and measure the time taken to compute the product matrix using the above three methods. Compare the running times to justify the above discussion. You must pick matrices of appropriate size to see the desired results. We have already provided you with a basic kickstarter code *matmult_dnc.c* to help you implement the divide-and-conquer approach that Strassen's method necessitates.

*[For further reading, please refer to Section 4.2 of Chapter 1 from your reference book (*Cormen T.H., Leiserson, C.E., Rivest, R.L., and C. Stein. *Introduction to Algorithms*, MIT Press, 3rd Edition, 2009*).]*