Nested Iterations or Nested Loops

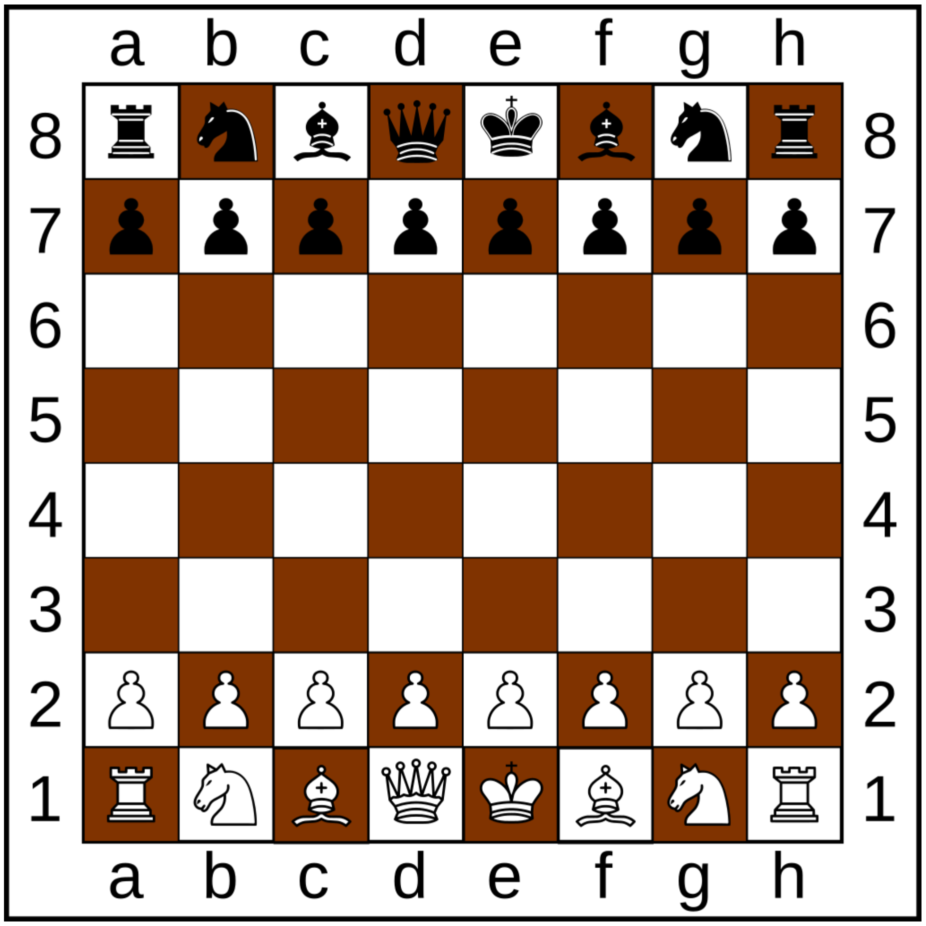

Figure 3: Chessboard[1]

If you were to print the reference to all the cells of a chess board, how are we going to do it? For this problem, instead of using the letters from a to h, the numbers from 1 to 8 will be utilized. So when printing cell a3, print 1-3 instead. 7-5 instead of g-5. The resulting printed cell references should look like this:

[1]Image taken from https://commons.wikimedia.org/wiki/File:AAA_SVG_Chessboard_and_chess_pieces_02.svg

```
1-8 2-8 3-8 4-8 5-8 6-8 7-8 8-8
1-7 2-7 3-7 4-7 5-7 6-7 7-7 8-7
1-6 2-6 3-6 4-6 5-6 6-6 7-6 8-6
1-5 2-5 3-5 4-5 5-5 6-5 7-5 8-5
1-4 2-4 3-4 4-4 5-4 6-4 7-4 8-4
1-3 2-3 3-3 4-3 5-3 6-3 7-3 8-3
1-2 2-2 3-2 4-2 5-2 6-2 7-2 8-2
1-1 2-1 3-1 4-1 5-1 6-1 7-1 8-1
```

For each line that you see above, notice that for each pair, the first number starts at 1 and it is incremented all the way to 8 (the are 8 columns on a chess board). The second number is remains the same, 8 (there 8 rows on a chess board) in the case of the first line. As you go down the lines, this second number gets decremented. And when you reach the $8th$ line, it has be reduced to 1. For the first number, we observe the same pattern for all the lines.

Let's do this step by step. We will begin with printing just the row and we know this involves repetition. We need some variable that goes from 1 to 8. That is easy.

```c
#include <stdio.h>

int main(){
    int i;

    for(i=1; i<=8; i++)
        printf("%i-8 ", i);

    printf("\n");
    return 0;
}
```

We can repeat this loop 8 times and simply change 8 to 7, then 6, and all the way down to 1 as shown below.

```c
#include <stdio.h>

int main(){
    int i;

    for(i=1; i<=8; i++)
        printf("%i-8 ", i);

    for(i=1; i<=8; i++)
        printf("%i-7 ", i);

    for(i=1; i<=8; i++)
        printf("%i-6 ", i);

    for(i=1; i<=8; i++)
        printf("%i-5 ", i);

    for(i=1; i<=8; i++)
        printf("%i-4 ", i);

    for(i=1; i<=8; i++)
        printf("%i-3 ", i);

    for(i=1; i<=8; i++)
        printf("%i-2 ", i);

    for(i=1; i<=8; i++)
        printf("%i-1 ", i);

    printf("\n");
    return 0;
}
```

Actually, we could have just printed everything using just `printf` statements and no loops at all! One `printf` statement for each line of output that looks like this: `printf("1-8 2-8 3-8 4-8 5-8 6-8 7-8 8-8\n");`.

But what if the board is variable where its size can go from $2x2$ all the way to some nxn. And to force this, let's ask the board size from the user.

```c
#include <stdio.h>

int main(){
    int i, boardSize;

    do{
        printf("Enter the size of the board: ");
        scanf("%i",&boardSize);
    }while(boardSize < 2);

    for(i=1; i <= boardSize; i++)
        printf("%i-%i ",i,boardSize);

    printf("\n");
    return 0;
}
```

Because *boardSize* is variable, there is no way we can predict how many copies of the `for` statement we have to make, unlike when the board was set at $8x8$, the size of a chess board.

What we need to then is look at the *boardSize*. To successfully print the desired result, not only should i change value, but *boardSize* as well. It decrements line after line. And this means repetition as well.

```c
#include <stdio.h>

int main(){
    int i, boardSize, temp;

    do{
        printf("Enter the size of the board: ");
        scanf("%i",&boardSize);
    }while(boardSize < 2);

    temp = boardSize;

    for(i=1; i <= boardSize; i++){
        printf("%i-%i ",i,temp);
        temp--;
    }

    printf("\n");
    return 0;
}
```

Instead of directly changing `boardSize`, we used a temporary storage for it. Compile the source above and run it. We didn't quite get the result we desired: `1-8 2-7 3-6 4-5 5-4 6-3 7-2 7-1`.

Observe that while the iterator i increments, *temp* should not change just yet. Only after each line should *temp* decrement. This suggests that for every *temp* value, i goes from to *boardSize*. And as a value for *temp* is obtained i restarts from 1. AWhat we have described here is a loop within a loop or nested loop.

Check the simple nested loop below.

```c
#include <stdio.h>

int main(){
    int i,j;

    for(i=1; i <= 3; i++){
        for(j=1; j <= 3; j++)
            printf("%i-%i ",i,j);
        printf("\n");
    }

    return 0;
}
```

We introduced a j to be used in the second `for` statement. observe that the second `for` statement is different from the case where you have 2 for statements where the second comes after the first as shown below.

```c
#include <stdio.h>

int main(){
    int i,j;

    for(i=1; i <= 3; i++)
        printf("%i ", i);
    for(j=1; j <= 3; j++)
        printf("%i  ",j);

    printf("\n");

    return 0;
}
```

The second loop will only start executing once the first loop is done executing.

In the case of the nested loop above, for each iteration of the outer loop, an inner loop has to be executed and the `printf("\n")` statement. When i is 1, after the condition check, control of the execution will go the inner loop. This inner loop will then have to finish execution before executing the `printf("\n")` statement. And that is exactly why there's a line of output prints i with the consistent value 1 and j goes from 1 to 3: `1-1 1-2 1-3`. The control of the execution is then brought to the iterator of the outer loop `i++`. Since i is still less than 3, the inner loop will then be executed. And this cycle repeats for one last time when i is 3. And we get the result below.

```
1-1 1-2 1-3
2-1 2-2 2-3
3-1 3-2 3-3
```

Check the code below. Compile and run it and see if successfully prints the desired cell resferences of the chess board.

```
#include <stdio.h>

int main(){
    int i, boardSize, j;

    do{
        printf("Enter the size of the board: ");
        scanf("%i",&boardSize);
    }while(boardSize < 2);

    for(i=1; i <= boardSize; i++){
        for(j=boardSize; j>=1; j--)
            printf("%i-%i ",i,j);
        printf("\n");
    }

    return 0;
}
```

When boardSize is 8, we get the output below.

```
1-8 1-7 1-6 1-5 1-4 1-3 1-2 1-1
2-8 2-7 2-6 2-5 2-4 2-3 2-2 2-1
3-8 3-7 3-6 3-5 3-4 3-3 3-2 3-1
4-8 4-7 4-6 4-5 4-4 4-3 4-2 4-1
5-8 5-7 5-6 5-5 5-4 5-3 5-2 5-1
6-8 6-7 6-6 6-5 6-4 6-3 6-2 6-1
7-8 7-7 7-6 7-5 7-4 7-3 7-2 7-1
8-8 8-7 8-6 8-5 8-4 8-3 8-2 8-1
```

This is not quite the result we want. Notice that for the chess board, it is the second number of the pair that should remain constant for one line of output. And it should begin in 8. This tells us that it should be the outer loop and the inner loop.

```c
#include <stdio.h>

int main(){
    int i, boardSize, j;

    do{
        printf("Enter the size of the board: ");
        scanf("%i",&boardSize);
    }while(boardSize < 2);

    for(i=boardSize; i >= 1; i--){
        for(j=1; j <= boardSize; j++)
            printf("%i-%i ",j,i);
        printf("\n");
    }

    return 0;
}
```

It should now display the desired result.

```
1-8 2-8 3-8 4-8 5-8 6-8 7-8 8-8
1-7 2-7 3-7 4-7 5-7 6-7 7-7 8-7
1-6 2-6 3-6 4-6 5-6 6-6 7-6 8-6
1-5 2-5 3-5 4-5 5-5 6-5 7-5 8-5
1-4 2-4 3-4 4-4 5-4 6-4 7-4 8-4
1-3 2-3 3-3 4-3 5-3 6-3 7-3 8-3
1-2 2-2 3-2 4-2 5-2 6-2 7-2 8-2
1-1 2-1 3-1 4-1 5-1 6-1 7-1 8-1
```

Power problem without using the multiplication operator

Recall the power the problem.

```c
#include <stdio.h>

int main(){
    int base, exp, iter = 1, ans = 1;

    printf("Enter the upper bound: ");
    scanf("%i%i",&base,&exp);

    while(iter <= exp){
        ans = ans * base;
        iter++;
    }

    printf("%i^%i is %i.\n",base,exp,ans);
    return 0;
}
```

Recall as well multiplication is essentially repetitive addition as explained in Problem Solving.

```c
#include <stdio.h>

int main(){
    int x, y, iter = 1, prod = 0;

    printf("Enter the factors: ");
    scanf("%i%i",&x,&y);

    while(iter <= y){
        sum = sum + x;
        iter++;
    }

    printf("%i * %i is %i.\n",x,y,sum);
    return 0;
}
```

This means that we will do this repetitive addition in the power computation instead of the the multiplication operator. But this needs careful analysis. Let's zoom in to the repetitive multiplication.

```
while(iter <= exp){
    ans = ans * base;
    iter++;
}
```

What needs to be converted into repetitive addition is `ans = ans * base`. Let's perform `ans * base`. We set `x` to `ans` and `y` to base. This means we need a second iterator, and let's call it iterAdd. We also need a variable to store the result of the repetitive addition, and let's call it prod (since it really represents the product).

```
iterAdd = 1;
prod = 0;
x = ans;
y = base;

while(iterAdd <= y){
    prod = prod + x;
    iterAdd++;
}
```

This entire repetitive addition code for multplication will then have to repeated exponent times.

```c
#include <stdio.h>

int main(){
    int base, exp, iter = 1, ans = 1, iter2, x, y, sum;

    printf("Enter the upper bound: ");
    scanf("%i%i",&base,&exp);

    while(iter <= exp){
        sum = 0;
        x = ans;
        y = base;
        iter2 = 1;
        while(iter2 <= y){
            sum = sum + x;
            iter2++;
        }
        iter++;
        ans = sum;
    }
    printf("%i^%i is %i.\n",base,exp,ans);
    return 0;
}
```

Head on the video lecture about this for a more detailed explanation.

Prime numbers less than n

Recall the prime numbers less than 10 problem? What if we generalize it to displaying all prime numbers less than some positive number n? Here, we are given a number n, say, 15. What are the prime numbers less than 15? These are 2, 3, 5, 7, 11, and 13.

The easiest solution to this problem is to go from 2 (smallest prime) all the way to 14 (n is not included since the problem specifies only those less than n) while checking if any of these numbers are prime. We already know how to check if a certain number is prime or not. We also know how to generate numbers from some lower bound l to some upper bound n. Knowing these, it should now be easy to solve this problem.

Generating the numbers from 2 to $n-1$ (let's call them cp, short for candidate primes) easily look like this:

```c
for(cp = 2; cp<n; cp++) //cp is short for candidate prime
```

Then for each of these candidate primes, we need to verify their primality. Again, this is easy because we did this already. Check it (down) below.

```
prime = 1;

for(cf = 2; prime==1 && cf*cf <= cp; cf++){ //notice the
↪   use of compound condition
    if(cp%cf==0)
        prime = 0; //the break is not used because of the
↪   compound condition
}
```

Recall that the candidate factors only go all the way until the square-root of n only. And recall that *prime* is used as a flag. Since it is initially 1, a non-zero number, we assume that *cp* is prime. Once we find a candidate factor that evenly divides *cp*, then *cp* is not prime. Hence, setting *prime* to 0. Recall 0 means false.

Now that these are setup. There is only one thing left to do. To print cp if it is prime. The solution should look like this:

```
#include <stdio.h>

int main(){
    int n, cp, cf, prime;

    do{
        printf("Enter a positive n: ");
        scanf("%i",&n);
    }while(n < 1);

    for(cp = 2; cp < n; cp++){  //cp is short for candidate
     ↪   prime
        prime = 1;              // we assume that cp is
↪   prime and
                                //prime gets reset for each
                                 ↪   cp generated
        for(int cf = 2; prime == 1 && cf*cf <= cp ; cf++){
            if(cp%cf == 0)      //checks if cf is a factor
             ↪   of cp
                prime = 0;      //if cf divides cp, then cp
↪   is not prime
        }
        if(prime)               //if cp is prime, print it
            printf("%i ",cp);
    }

    printf("\n");
    return 0;
}
```

If only the prime numbers have a general or overall pattern, there wouldn't have been any need to check for the primality of each of the generated candidate primes. The problem with out solution is that it probably will take a while before successfully printing all the prime numbers less than a very large number. Run the program and test 10000000 as an input. Then try 100000000. Then finally, 1000000000. By the way, do not forget to comment out the `if(prime) printf("%i ",cp)` statement. Printing on the standard output takes a time.

Can we do better? The next section will reveal an answer towards the end.