CMSC 142 T2L — Project Milestone 0
Github Repository: https://github.com/jgbanaag/CMSC-142-Project

Members:
Almira, Anton
Alviar, John Angelo
Banaag, Jullianne
Pendon, Rainier
Villamin, Jan Neal Isaac

We are given first the permutation code that uses `N + 1` stacks and backtracking.

```
#include <stdio.h>

int main() {
      int N;
      printf("Enter N: ");
      scanf("%d", &N);
      int nopts[N + 2], option[N + 2][N + 2], start, move, i, 
candidate;
      move = start = 0;
      nopts[start] = 1;
      while (nopts[start] > 0) {
            if (nopts[move] > 0) {
                  nopts[++move] = 0;
                  if (move > N)
                        for (i = 1; i < move; i++) {
                              printf("%d", option[i][nopts[i]]);
                              printf(i < move - 1 ? " " : "\n");
                        }
                  for (candidate = N; candidate >= 1; candidate--) {
                        for (i = move - 1; i >= 1; i--)
                              if (option[i][nopts[i]] == candidate)
                                    break;
                        if (i == 0)
                              option[move][++nopts[move]] = candidate;
                  }
            } else
                  nopts[--move]--;
      }
      return 0;
 }
```

Here, we simply fill up candidates that are not taken as top of stacks and print whenever we get to the (`N + 1`)-th stack and backtrack whenever there are no more elements in the current stack.

To implement the combination code, we only need to tweak some parts of the code. Note that for combination, order does not matter, therefore, it is a good idea to maintain a single order out of all permutations of a subset. The order that seems to be the most intuitive and simple is the increasing order. Thus, from this part of the code:

```
for (candidate = N; candidate >= 1; candidate--) {
     for (i = move - 1; i >= 1; i--)
          if (option[i][nopts[i]] == candidate)
               break;
     if (i == 0)
          option[move][++nopts[move]] = candidate;
}
```

We change the lines:

```
     for (i = move - 1; i >= 1; i--)
          if (option[i][nopts[i]] == candidate)
               break;
     if (i == 0)
          option[move][++nopts[move]] = candidate;
```

To be:

```
 if (move == 1 || candidate > option[move - 1][nopts[move - 1]])
      option[move][++nopts[move]] = candidate;
 else
      break;
```

This means that in order for us to add the candidate in the stack, it must follow that we are either operating under the first stack (so all candidates are possible) (`move == 1`) or that the previous top of stack (which is currently part of the combination sequence) is less than the current candidate (`candidate > option[move - 1][nopts[move - 1]]`). This maintains that the subset sequence appears in increasing order. Note that comparison to the previous suffices, since by induction, all previous elements in the combination sequence currently are all less than or equal to the previous top of stack.

Then, instead of printing the sequence everytime we get to the (`N + 1`)-th stack, we print before we backtrack. The reason for this is that everytime we backtrack, we transition from a combination sequence to another since we are discarding an element and that sequence will never be scanned again. Hence, we have the following `else` block.

```
else {
      // print subset sequence
      for (i = 1; i < move; i++) {
            printf("%d", option[i][nopts[i]]);
            printf(i < move - 1 ? " " : "\n");
      }
      // backtrack
      nopts[--move]--;
}
```

Thus, we always get an increasing subset sequence of `1` to `N` which we print for each transition. Hence, the final code will be as follows.

```
#include <stdio.h>

int main() {
    int N;
    printf("N: ");
    scanf("%d", &N);
    int nopts[N + 2], option[N + 2][N + 2], start, move, i, j,
candidate;
    move = start = 0;
    nopts[start] = 1;
    while (nopts[start] > 0) {
        if (nopts[move] > 0) {
            nopts[++move] = 0;
            for (candidate = N; candidate >= 1; candidate--)
                if (move == 1 || candidate > option[move -
1][nopts[move - 1]])
                    option[move][++nopts[move]] = candidate;
                else
                    break;
        } else {
            for (i = 1; i < move; i++) {
                printf("%d", option[i][nopts[i]]);
                printf(i < move - 1 ? " " : "\n");
            }
            nopts[--move]--;
        }
    }
    return 0;
}
```

Here, the `if`-block represents filling up the next stack with candidates while the `else`-block represents backtracking. The reason for `1`-indexing and sizes `N + 2` is to have useful sentinel values such as `nopts[0] == 1`. The reason for `nopts[++move] = 0` is to initialize the current top of stack which we will increment if a candidate is found.

If `N = 4`, the output will be:

```
1 2 3 4
1 2 3
1 2 4
1 2
1 3 4
1 3
1 4
1
2 3 4
2 3
2 4
2
3 4
3
4
```

We trace the code for all possible combinations of integers from `1` to `N = 4`, by tracing the following: `option`, `nopts`, `move`, and printed sequence (if any) during each iteration of the main loop.

Note that `nopts[i]` represents the index (1-indexing) of the top of the `i`-th stack and `option[i][j]` represents the `j`-th element of the `i`-th stack. Therefore, `option[i][nopts[i]]` represents the top of the `i`-th stack. Next, `move` represents the current stack from `0` to `N + 1 = 5`. The goal of `move` is to build the sequence, `option` is to keep track of possible options, and `nopts` is to keep track of the currently chosen elements in the sequence.

Initially, the `option` table is empty, and the `0`-th top of stack is initialized as `1` (`nopts[0] = 1`) and the current stack is set to `0` (`move = 0`) to start the loop. We run the main loop until the `0`-th top of stack is `1` (`nopts[0] > 0`). This will only happen when we backtracked from the first stack, meaning there are no more sequences.

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]						
[2]						
[3]						
[4]						
[5]						

nopts

1					

move = 0

In the first iteration, since the current top of stack index is `1` (`nopts[0] > 0`), we will try to fill up the candidates of the next stack. But first, increment the stack pointer and initialize the current top of stack as `0` (`nopts[++move] = 0`).

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						

[1]						
[2]						
[3]						
[4]						
[5]						

nopts	1	0				

move = 1

Then, fill up candidates. Since the current stack is `1` (`move == 1`) (`if (move == 1 || ...)` is always true), we add all possible candidates from `N` to `1` (`for (candidate = N; candidate >= 1; candidate--)`). While we fill up the candidates, we also increment the top of stack (`++nopts[move]`). Set the current top of stack as the current candidate (`option[move][++nopts[move]] = candidate`). Then, go again for the next iteration.

---

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4				
[2]						
[3]						
[4]						
[5]						

nopts	1	1				

move = 1

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4				
[2]		3				
[3]						
[4]						
[5]						

nopts

nopts	1	2				

move = 1

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4				
[2]		3				
[3]		2				
[4]						
[5]						

nopts	1	3				

move = 1

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4				
[2]		3				
[3]		2				
[4]		**1**				
[5]						

nopts

1	4				

move = 1

Now, move to the next iteration. Do the same process, since the current top of stack index is more than `0` (`nopts[1] == 4 > 0`), we move to the `2`nd stack (`++move`) and initialize the top of stack index to be `0` (`nopts[++move] = 0`).

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4				
[2]		3				
[3]		2				
[4]		**1**				

[5]						

nopts	1	4	0			

move = 2

For the next stack, since the current stack is the `2nd` stack (`move == 2`) is not `1` but the previous top of stack is `1` (`option[move - 1][nopts[move - 1]] == 1`) (so, `if (move == 1 || candidate > option[move - 1][nopts[move - 1])` is true if `candidate > 1`), we will only consider all candidates from `N to 2`.

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4			
[2]		3				
[3]		2				
[4]		**1**				
[5]						

nopts	1	4	1			

move = 2

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						

[1]		4	4			
[2]		3	3			
[3]		2				
[4]		**1**				
[5]						

nopts	1	4	2			

move = 2

---

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4			
[2]		3	3			
[3]		2	**2**			
[4]		**1**				
[5]						

nopts	1	4	3			

move = 2

Candidate 1 is not greater than the previous top which is 1, so break the candidate acceptance. Go to the next stack.

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4		
[2]		3	3			
[3]		2	**2**			
[4]		**1**				
[5]						

nopts

1	4	3	1	0	0

move = 3

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4		
[2]		3	3	**3**		
[3]		2	**2**			
[4]		**1**				
[5]						

nopts

1	4	3	2	0	0

move = 3

Candidate 2 is not greater than the previous top which is 2, so break the candidate acceptance. Go to the next stack.

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	**4**	
[2]		3	3	**3**		
[3]		2	**2**			
[4]		**1**				
[5]						

nopts

1	4	3	2	1	0

move = 4

Candidate 3 is not greater than the previous top which is 3, so break the candidate acceptance. Go to the next stack.

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	**4**	
[2]		3	3	**3**		
[3]		2	**2**			
[4]		**1**				
[5]						

nopts						
	1	4	3	2	1	0

move = 5

We keep doing this until we get to the 5th stack. Notice that for the 5th stack, the previous candidate is 4 (which is N) (option[move - 1][nopts[move - 1]] == 4, so if condition is always false) which means that there are no candidates possible from N to 1 that are greater than N.

Thus, we backtrack to the next iteration since nopts[5] == 0 (nopts[move] > 0 is false). Thus, we go to the other block. We first print the top of stacks in order:

```
for (i = 1; i < move; i++) {
     printf("%d", option[i][nopts[i]]);
     printf(i < move - 1 ? " " : "\n");
}
```

Then, we back track (nopts[--move]--). So, move becomes 4 and its top of stack index is now 0. Hence, we get our first combination: 1 2 3 4.

Printed: 1, 2, 3, 4

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		3	3	**3**		
[3]		2	**2**			
[4]		**1**				
[5]						

nopts						
	1	4	3	2	0	0

move = 4

Since the current top of stack (`nopts[4] == 0`), we go to the else block again and print the top of stacks in order and backtrack.

option

Printed: 1, 2, 3

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	**4**	4	
[2]		3	3	3		
[3]		2	**2**			
[4]		**1**				
[5]						

nopts	1	4	3	1	0	0

move = 3

After this, we will try again to go to the next stack and fill up candidates, but `option[move - 1][nopts[move - 1]] == option[3][1] == N == 4` which means no other candidate will be filled. Now, in the next iteration, we will backtrack since `nopts[move] == nopts[4] == 0`. Thus, we go to the `else`-block and we get the next combination.

option

Printed: 1, 2, 4

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		3	3	3		
[3]		2	**2**			

[4]		**1**				
[5]						

nopts	1	4	3	0	0	0

move = 3

Again, the current top of stack is `0 (nopts[3] == 0)`, so print and backtrack again.

---

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		3	**3**	3		
[3]		2	2			
[4]		**1**				
[5]						

Printed: 1, 2

nopts	1	4	2	0	0	0

move = 2

We keep doing this and the next traces can be found below. We fill possible candidates greater than `3`.

---

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	**4**	4	

[2]		3	**3**	3		
[3]		2	2			
[4]		**1**				
[5]						

nopts	1	4	2	1	0	0

move = 3

Go to the 4-th stack and fill possible candidates greater than the previous top which is `4`.

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	**4**	4	
[2]		3	**3**	3		
[3]		2	2			
[4]		**1**				
[5]						

nopts	1	4	2	1	0	0

move = 4

Go to else-block since the index of the top of the 4-th stack is `0`. Backtrack to the `3rd` stack.

---

option	[0]	[1]	[2]	[3]	[4]	[5]

Printed: 1, 3, 4

[0]						
[1]		4	4	4	4	
[2]		3	**3**	3		
[3]		2	2			
[4]		**1**				
[5]						

nopts	1	4	2	0	0	0

move = 3

Backtrack to the `2nd` stack since the current top of stack index is `0`.

Printed: 1, 3

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	**4**	4	4	
[2]		3	3	3		
[3]		2	2			
[4]		**1**				
[5]						

nopts	1	4	1	0	0	0

move = 2

Fill up candidates in the `3rd` stack that are greater than the previous top `4`.

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	**4**	4	4	
[2]		3	3	3		
[3]		2	2			
[4]		**1**				
[5]						

nopts

1	4	1	0	0	0

move = 3

Backtrack since the top of `3rd` stack index is `0`.

---

option

Printed: 1, 4

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		3	3	3		
[3]		2	2			
[4]		**1**				
[5]						

nopts					
1	4	0	0	0	0

move = 2

Backtrack again since the top of the 2nd stack index is 0.

option

Printed: 1

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		3	3	3		
[3]		**2**	2			
[4]		1				
[5]						

nopts					
1	3	0	0	0	0

move = 1

Try candidates for the 2nd stack that are greater than the previous top, 2.

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						

[1]		4	**4**	4	4	
[2]		3	3	3		
[3]		**2**	2			
[4]		1				
[5]						

nopts						
	1	3	1	0	0	0

move = 2

---

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		3	**3**	3		
[3]		**2**	2			
[4]		1				
[5]						

nopts						
	1	3	2	0	0	0

move = 2

Try candidates for the `3rd` stack that are greater than the previous top, `3`.

---

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	**4**	4	

[2]		3	**3**	3		
[3]		**2**	2			
[4]		1				
[5]						

nopts	1	3	2	1	0	0

move = 3

Try candidates for the `3rd` stack that are greater than the previous top, `4`.

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	**4**	4	
[2]		3	**3**	3		
[3]		**2**	2			
[4]		1				
[5]						

nopts	1	3	2	1	0	0

move = 4

Backtrack since the top of the `4`-th stack index is `0`.

option	[0]	[1]	[2]	[3]	[4]	[5]

Printed: 2, 3, 4

[0]						
[1]		4	4	4	4	
[2]		3	**3**	3		
[3]		**2**	2			
[4]		1				
[5]						

nopts	1	3	2	0	0	0

move = 3

Backtrack since the top of the `3rd` stack index is `0`.

Printed: 2, 3

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	**4**	4	4	
[2]		3	3	3		
[3]		**2**	2			
[4]		1				
[5]						

nopts	1	3	1	0	0	0

move = 2

Try all candidates of the `3rd` stack that are greater than the previous top, `4`.

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	**4**	4	4	
[2]		3	3	3		
[3]		**2**	2			
[4]		1				
[5]						

nopts

nopts	1	3	1	0	0	0

move = 3

Backtrack since the top of the `3rd` stack index is `0`.

---

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		3	3	3		
[3]		**2**	2			
[4]		1				
[5]						

Printed: 2, 4

nopts					
1	3	0	0	0	0

move = 2

Backtrack since the top of the `2nd` stack index is `0`.

option

Printed: 2

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		**3**	3	3		
[3]		2	2			
[4]		1				
[5]						

nopts

1	2	0	0	0	0

move = 1

Try all candidates of the `2nd` stack that are greater than the previous top of stack, `3`.

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	**4**	4	4	
[2]		**3**	3	3		
[3]		2	2			

[4]		1				
[5]						

nopts	1	2	1	0	0	0

move = 2

Try all candidates of the `3rd` stack that are greater than the previous top of stack, `4`.

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	**4**	4	4	
[2]		**3**	3	3		
[3]		2	2			
[4]		1				
[5]						

nopts	1	2	1	0	0	0

move = 3

Backtrack since the top of the `3rd` stack index is `0`.

Printed: 3, 4

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	

[2]		**3**	3	3		
[3]		2	2			
[4]		1				
[5]						

nopts	1	2	0	0	0	0

move = 2

Backtrack since the top of the `2nd` stack index is `0`.

---

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		**4**	4	4	4	
[2]		3	3	3		
[3]		2	2			
[4]		1				
[5]						

Printed: 3

nopts	1	1	0	0	0	0

move = 1

Try all candidates of the `2nd` stack that are greater than the previous top of stack, `4`.

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		**4**	4	4	4	
[2]		3	3	3		
[3]		2	2			
[4]		1				
[5]						

nopts	1	1	0	0	0	0

move = 2

Backtrack since the top of the `2nd` stack index is `0`.

---

option	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]		4	4	4	4	
[2]		3	3	3		
[3]		2	2			
[4]		1				
[5]						

Printed: 4

nopts	1	0	0	0	0	0

move = 1

Backtrack since the top of the `1st` stack index is `0`.

option

	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]						
[2]						
[3]						
[4]						
[5]						

Printed: none

nopts

0	0	0	0	0	0

move = 0

For the final iteration, since we backtracked to the `0-th` stack and `nopts[0]` became `0`, the main loop stopped.