This is Section.

Week 3: Algorithms

Gabe LeBlanc

Attendance Form: tinyurl.com/gabesection3

Questions before we begin?

- What are **structs?**
- How do we define and use our own **functions**?
- What is **Big *O* notation**?

Structs

```
string name = “Joe Biden”;
int votes = 10;
```

```
string name = “Joe Biden”;
int votes = 10;
```

```
??? president[2] = {“Joe Biden”, 10};
```

```
string name = “Joe Biden”;
int votes = 10

??? president[ = {    e Biden”, 10};
```

```
typedef struct
{
    string name;
    int votes;
}
candidate;
```

```
typedef struct
{
  string name;
  int votes;
}
candidate;
```

```
typedef struct
{
    string name;
    int votes;
}
candidate;
```

```
typedef struct
{
    string name;
    int votes;
}
candidate;
```

```
candidate president;
candidate president;
president.name = "Alyssa";
president.votes = 10;
```

```
candidate candidates[2];

candidates[0].name = “Gabe”;
candidates[0].votes = 3;

candidates[1].name = “Remy”;
candidates[1].votes = 10000000;
```

```
candidate candidates[2];

candidates[0].name = “Gabe
candidates[0].votes = 3;

candidates[1].name = “Re
candidates[1].votes = 10
```

```
candidate candidates[2];

candidates[0].name = “Gabe
candidates[0].votes = 3;

candidates[1].name = “Re
candidates[1].votes = 1
```

name	Alice	Bob	Charlie
votes	2	1	3

```
candidates[0];
```

name	Alice	Bob	Charlie
votes	2	1	3

```
candidates[0].name;
```

name	Alice	Bob	Charlie
votes	2	1	3

```
candidates[0].votes;
```

What are algorithms?

You’ve already been using them.

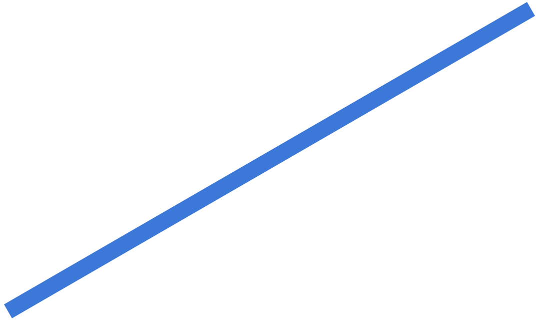
Runtime analysis

O(N) — "worst case" definition

In the worst case, I need to do approximately N steps for an input of size N.

O(N) — "scaling" definition

For every new item that gets added to my algorithm's input, the algorithm needs to do a fixed number of new steps. We say "our runtime scales **linearly** with the size of our input".

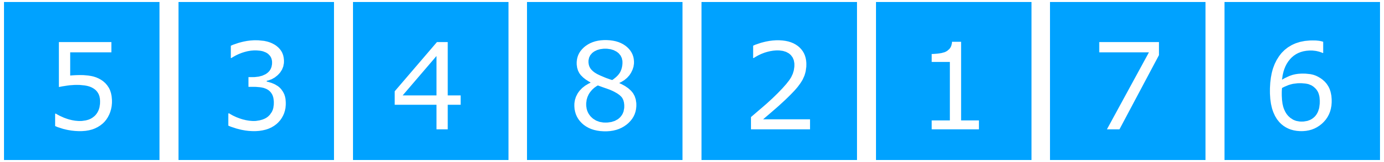


Ω(N) — "best case" definition

No matter the input size, I'll always have to do approximately n steps.

Sorting

Bubble Sort

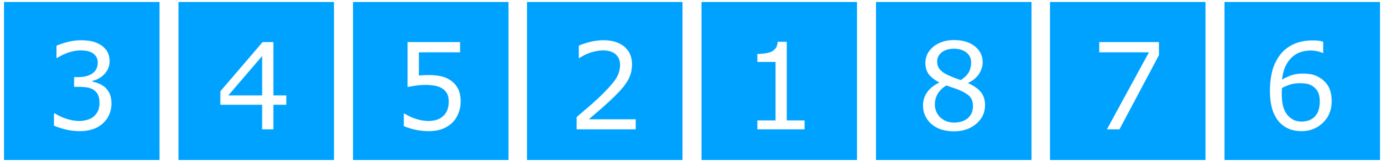


5	3	4	8	2	1	7	6

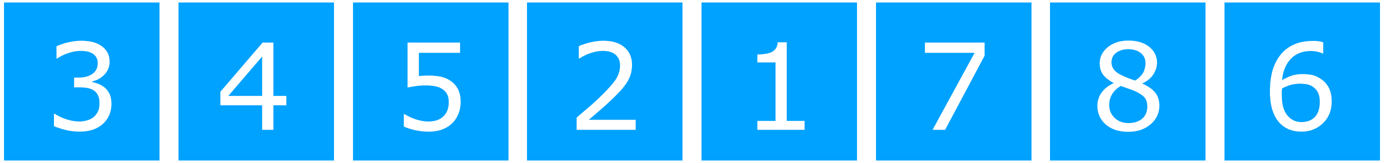
3	5	4	8	2	1	7	6

3	4	5	8	2	1	7	6

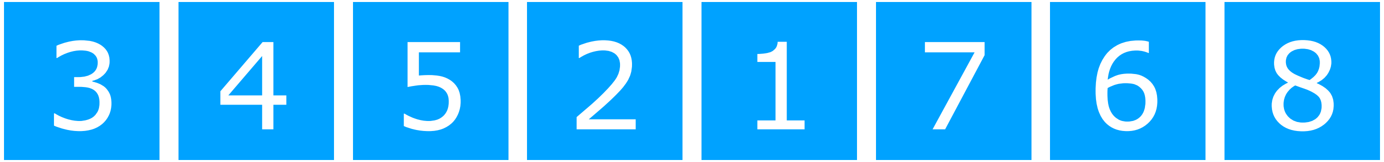
3	4	5	2	8	1	7	6



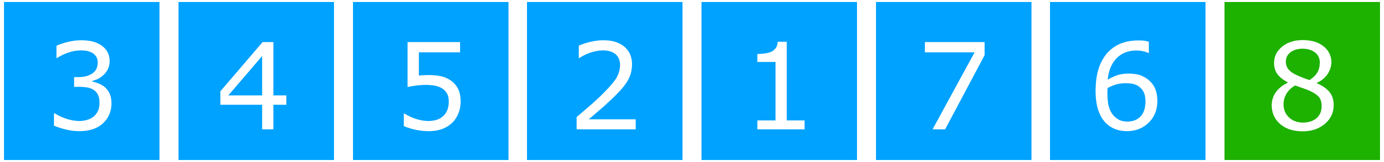
3 4 5 2 1 8 7 6



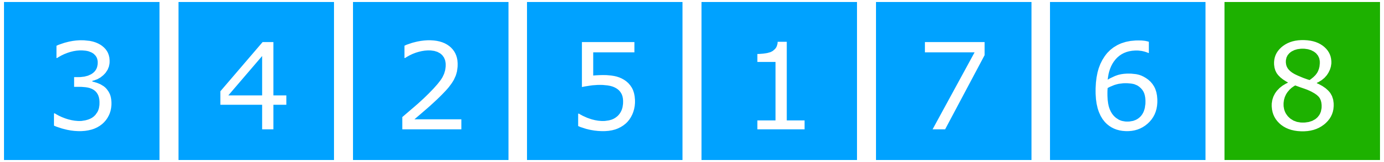
3 4 5 2 1 7 8 6



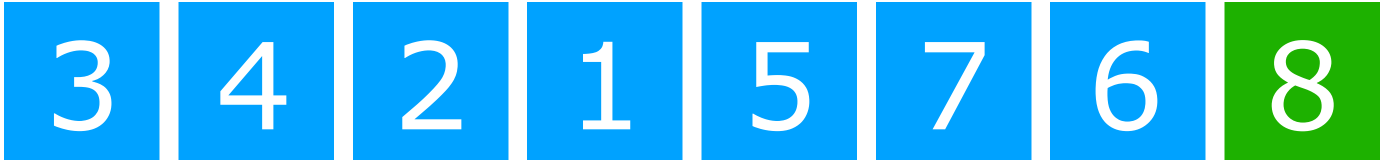
3 4 5 2 1 7 6 8



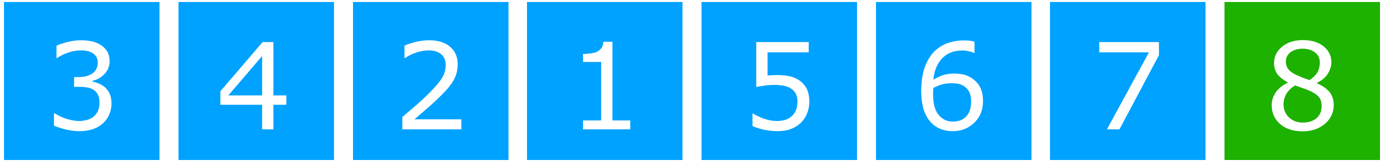
3 4 5 2 1 7 6 8



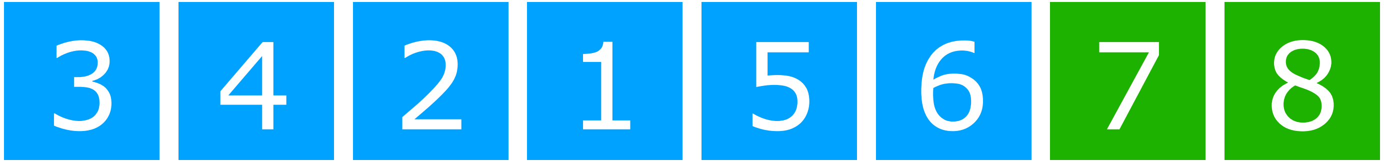
3	4	2	5	1	7	6	8



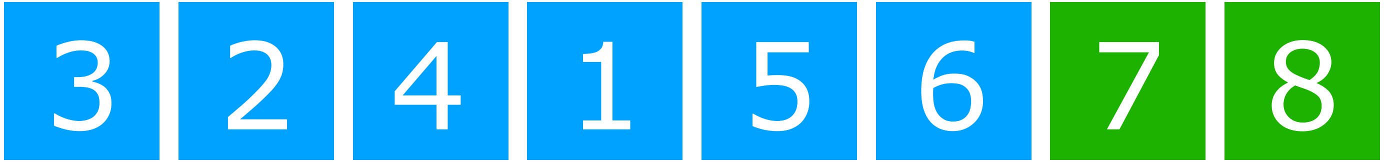
3 4 2 1 5 7 6 8



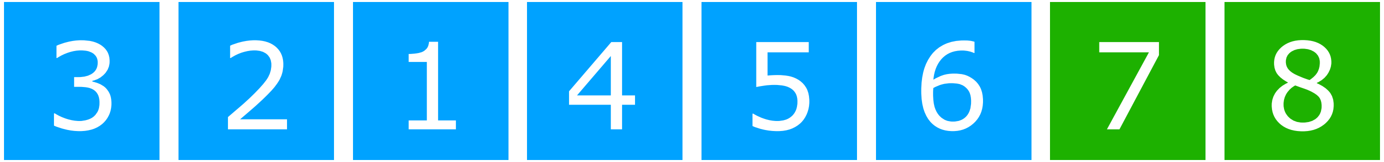
3	4	2	1	5	6	7	8



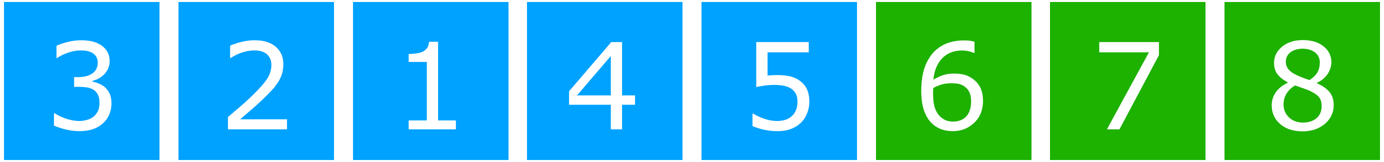
3	4	2	1	5	6	7	8



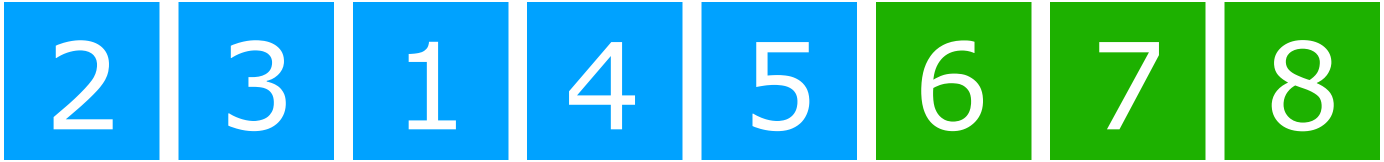
3	2	4	1	5	6	7	8



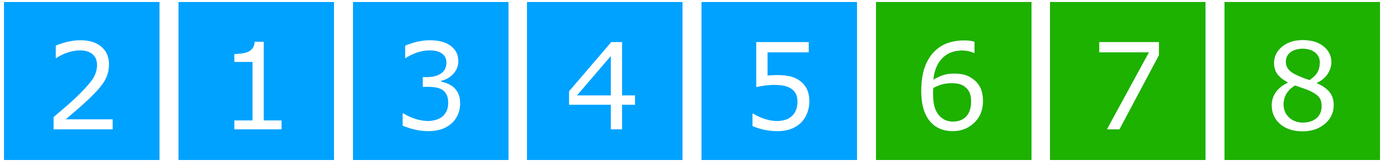
3 2 1 4 5 6 7 8



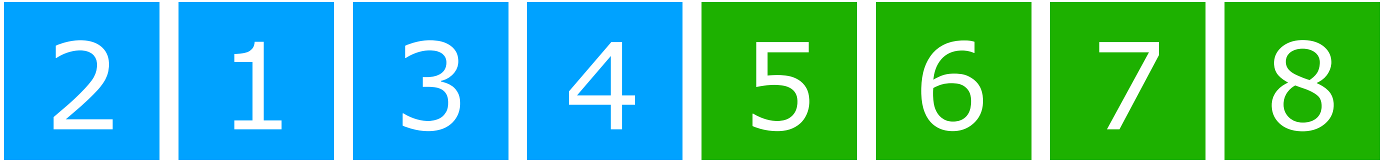
3 2 1 4 5 6 7 8



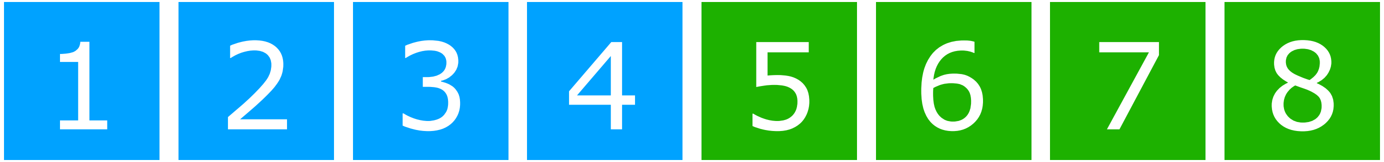
2 3 1 4 5 6 7 8



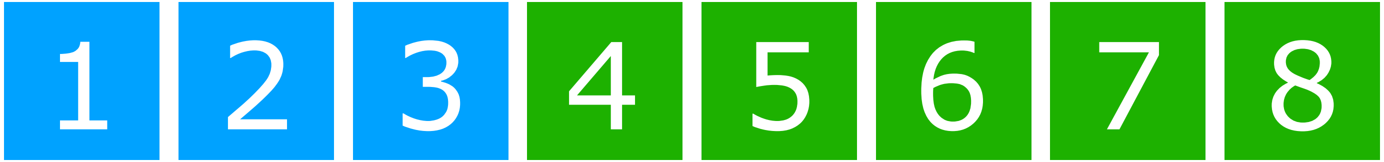
2	1	3	4	5	6	7	8



2	1	3	4	5	6	7	8



1 2 3 4 5 6 7 8



1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

```
Repeat for every element in our list, except last:
    Look at each element from first to second-to-last:
        If current and next elements out of order:
            Swap them
```

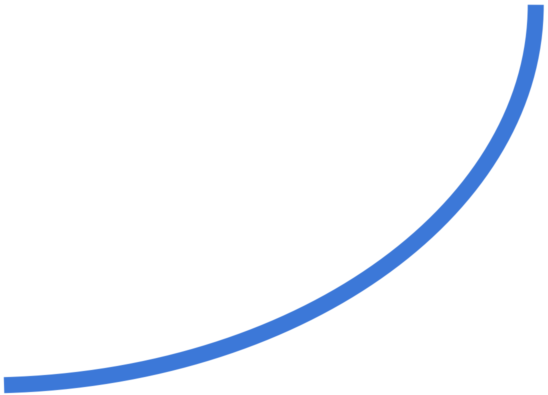
```
Repeat n - 1 times
    For j from 0 to n - 2
        If j'th and j + 1'th elements out of order
            Swap them
```

O(N^2) — "worst case" definition

In the worst case, I need to do approximately N^2 steps if my input size is N.

O(N^2) — "scaling" definition

For every new item that gets added to my input, I need to do approximately **N** new steps.

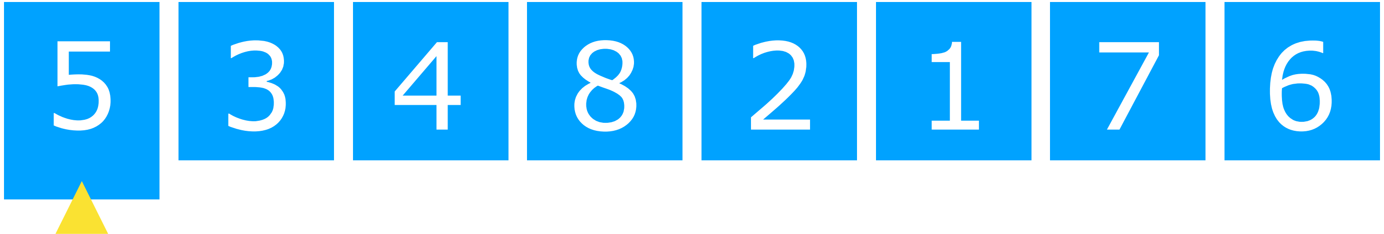


Ω(N) — "best case" definition

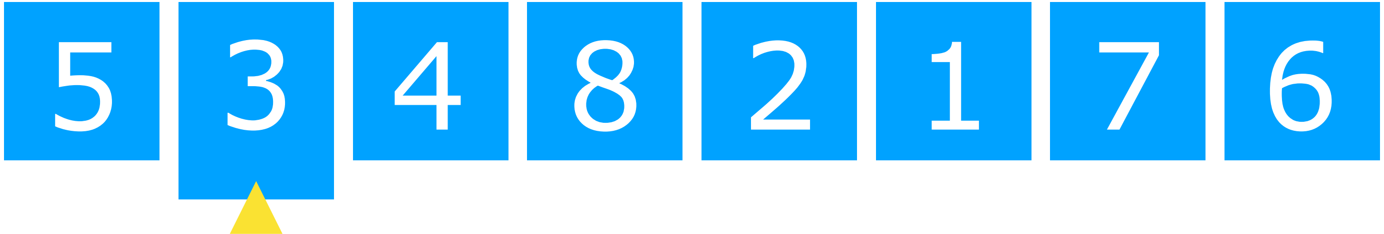
In the best case, I need to do approximately N steps if my input size is N.

Selection Sort

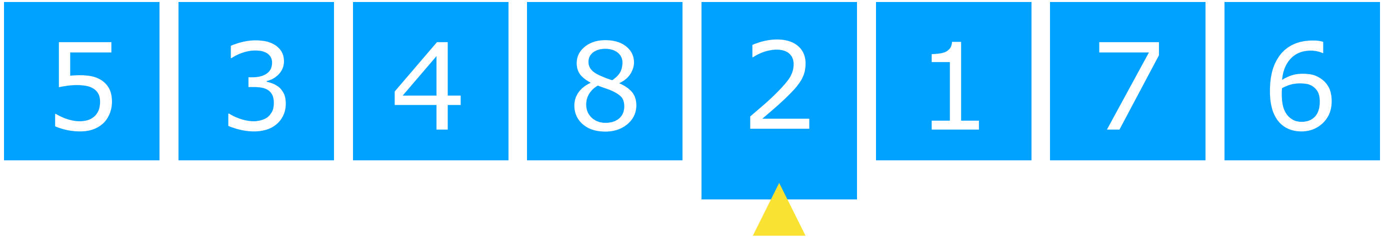
5	3	4	8	2	1	7	6



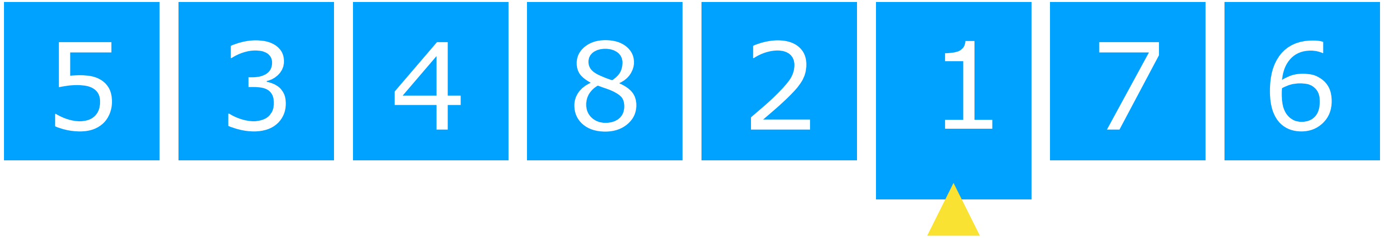
5 3 4 8 2 1 7 6



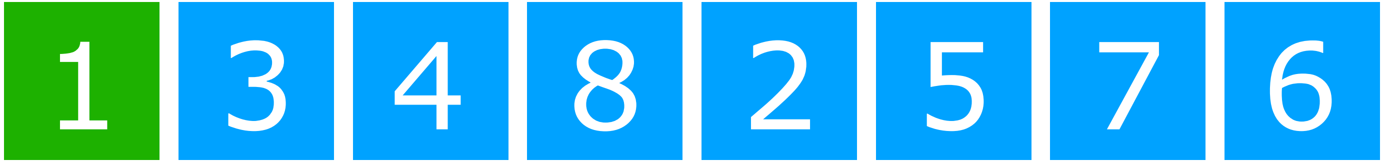
5	3	4	8	2	1	7	6



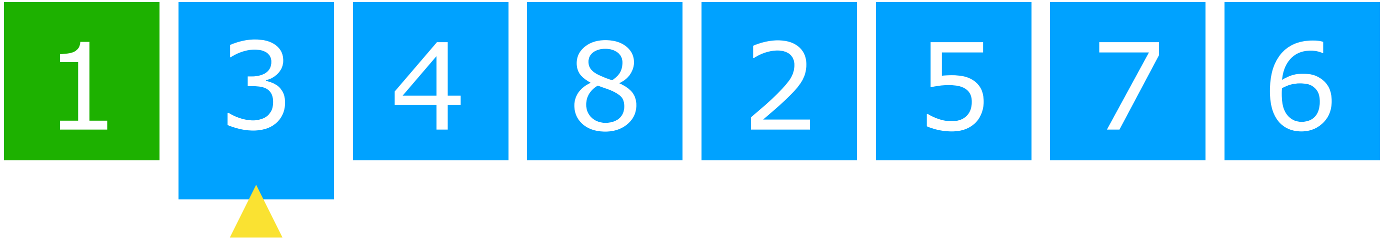
5 3 4 8 2 1 7 6



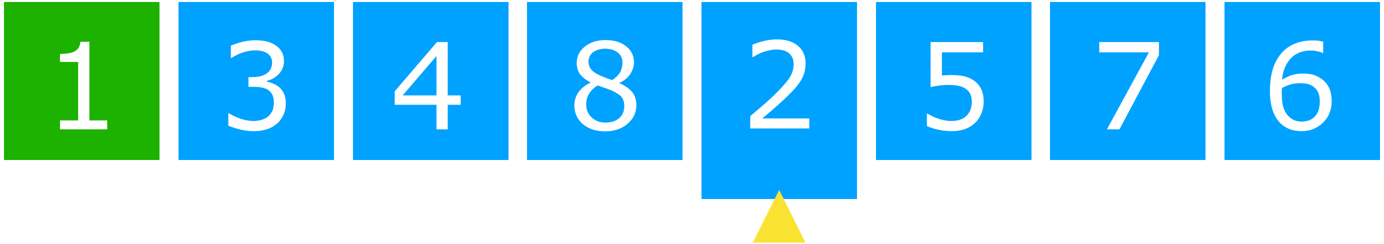
5 3 4 8 2 1 7 6



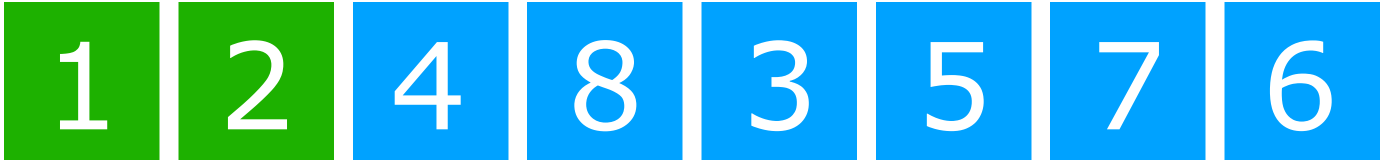
1	3	4	8	2	5	7	6



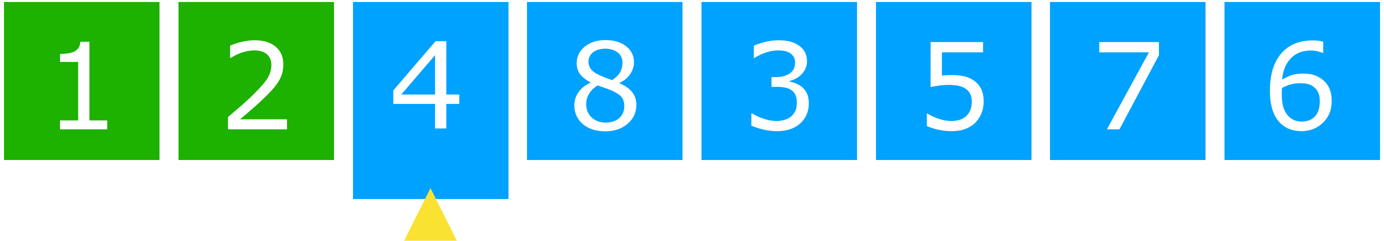
1 3 4 8 2 5 7 6



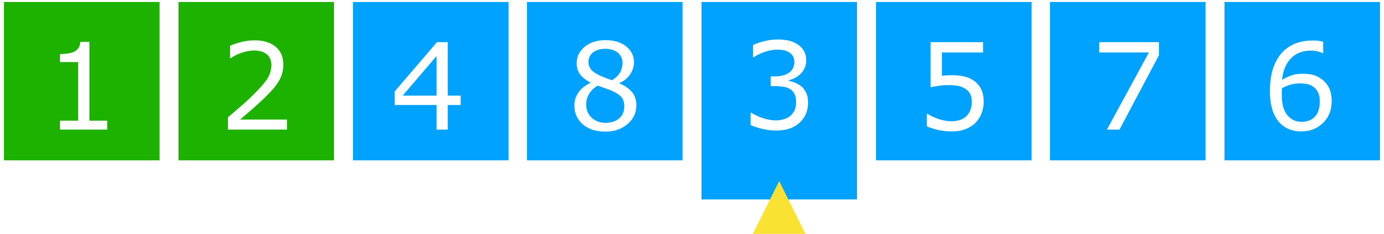
1	3	4	8	2	5	7	6

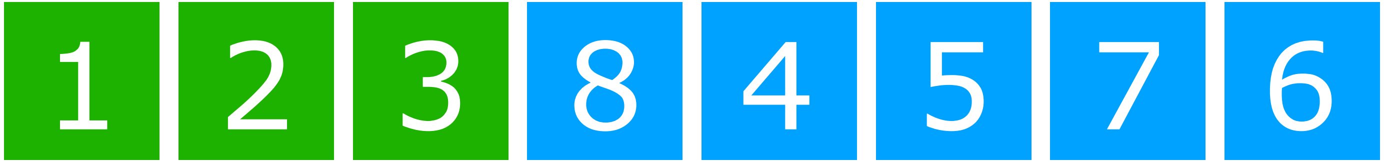


1 2 4 8 3 5 7 6

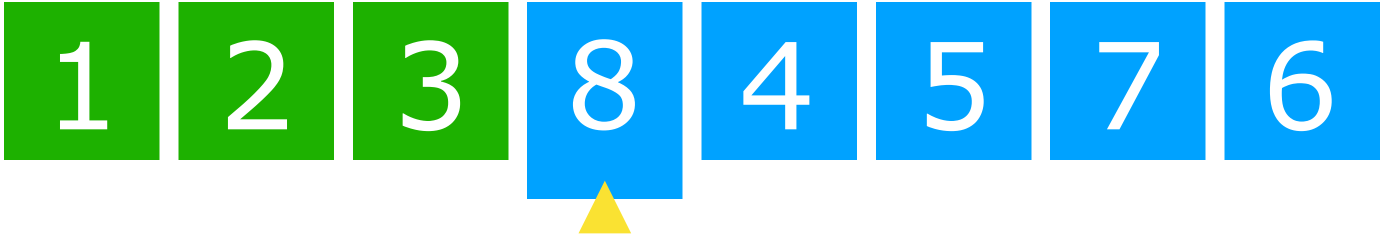


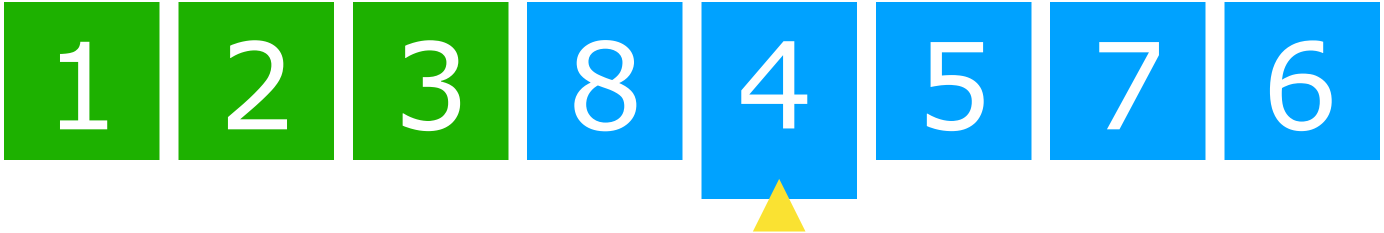
1 2 4 8 3 5 7 6


1
2
4
8
3
5
7
6

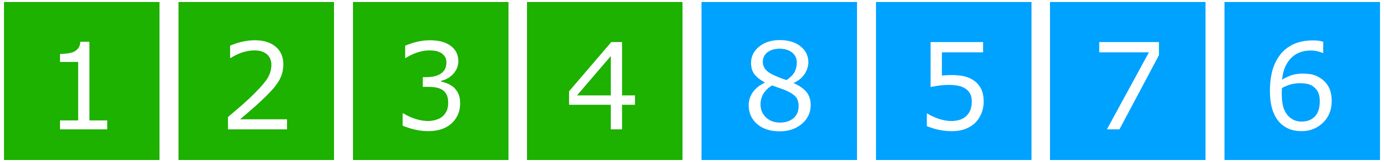


1	2	3	8	4	5	7	6

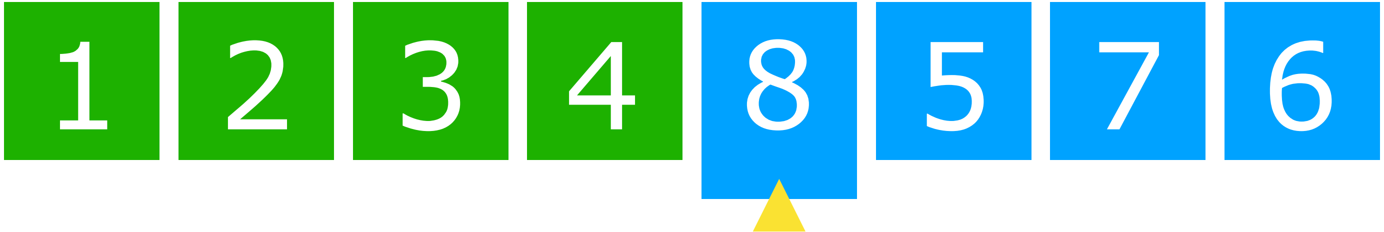

1
2
3
8
4
5
7
6



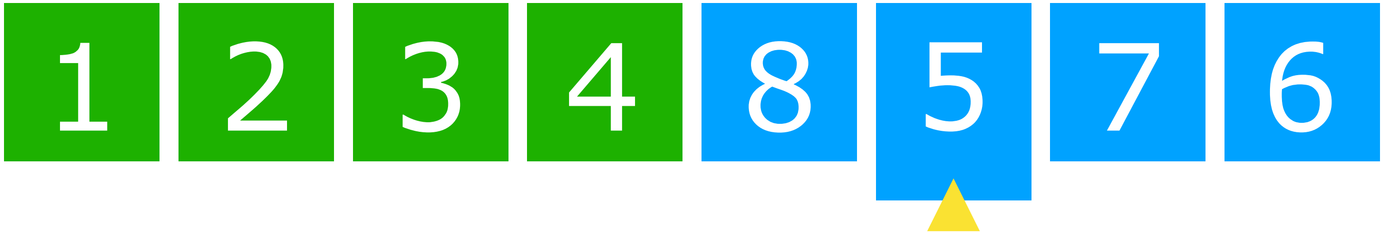
1 2 3 8 4 5 7 6



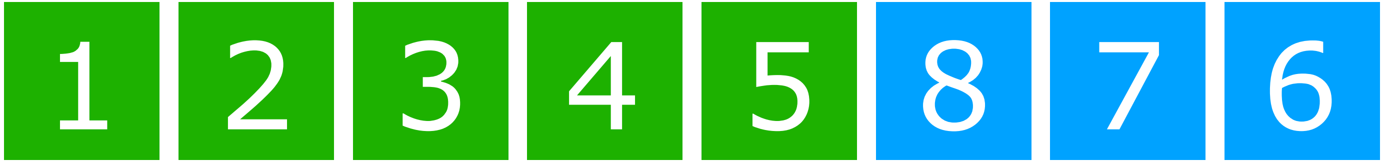
1	2	3	4	8	5	7	6



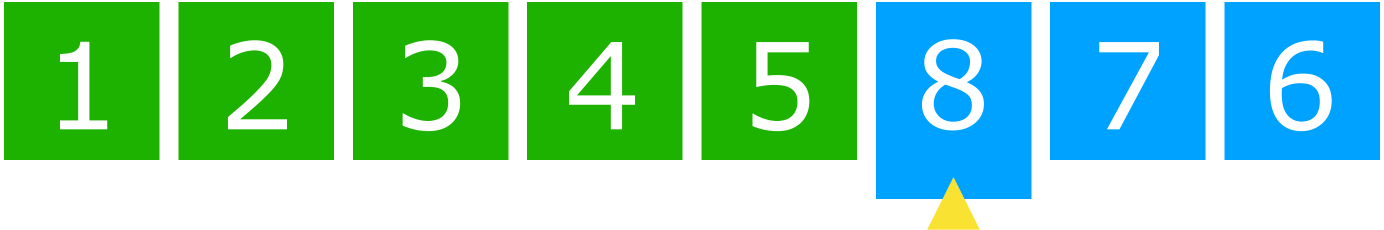
1 2 3 4 8 5 7 6



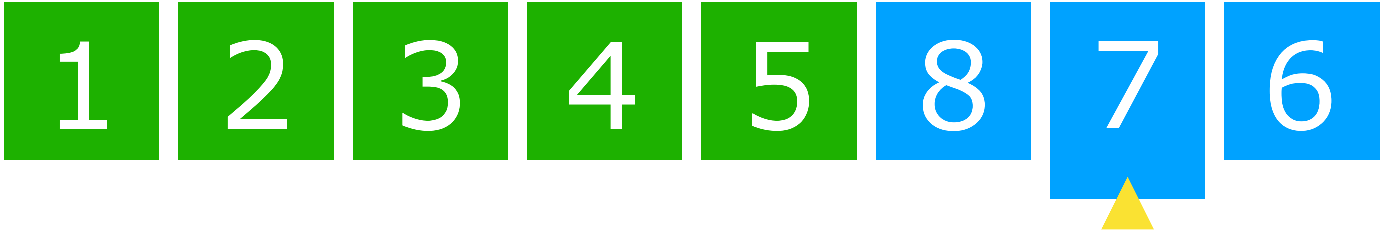
1 2 3 4 8 5 7 6

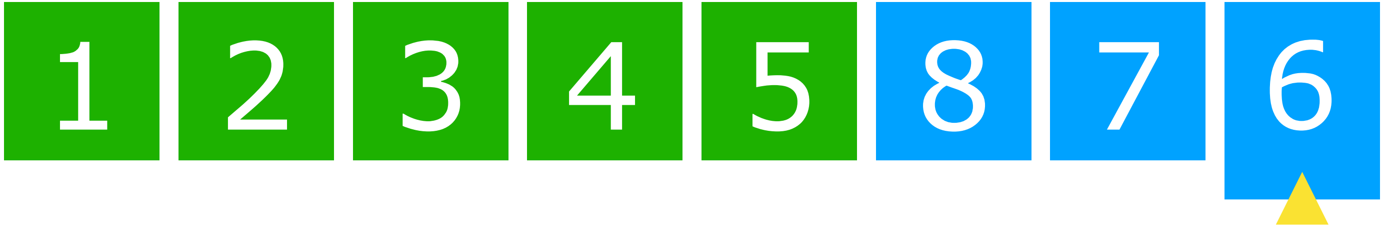


1 2 3 4 5 8 7 6

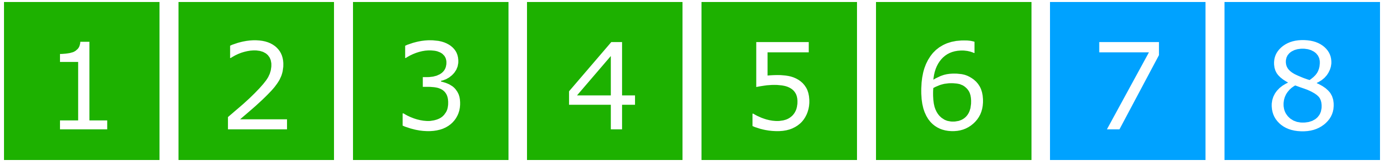


1 2 3 4 5 8 7 6

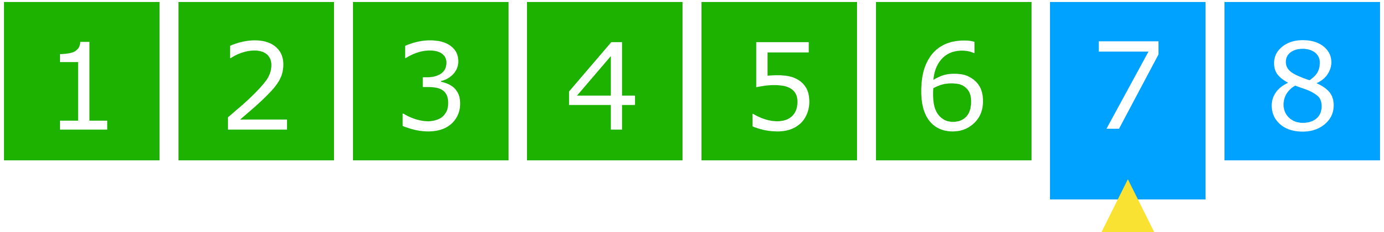

1
2
3
4
5
8
7
6

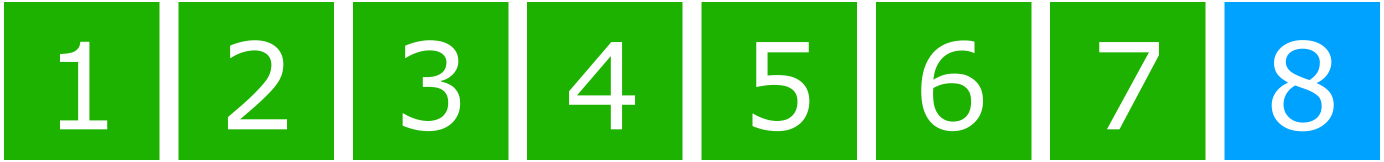


1 2 3 4 5 8 7 6

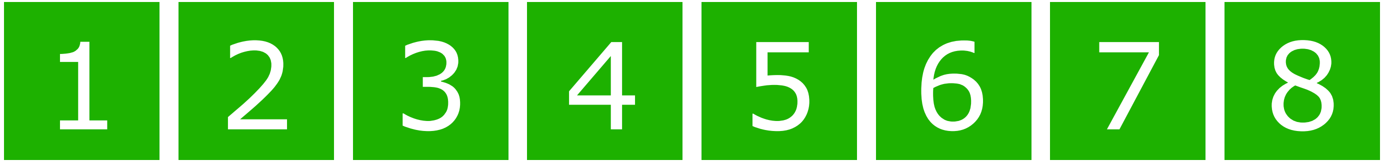


1 2 3 4 5 6 7 8


1
2
3
4
5
6
7
8



1 2 3 4 5 6 7 8



1 2 3 4 5 6 7 8

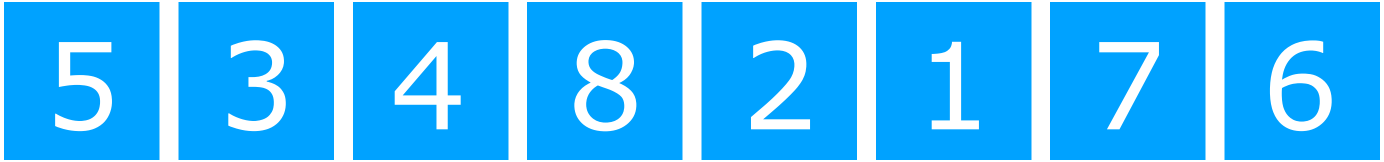
O(N^2) — "worst case" definition

In the worst case, I need to do approximately N^2 steps if my input size is N.

$\Omega(N^2)$ — "best case" definition

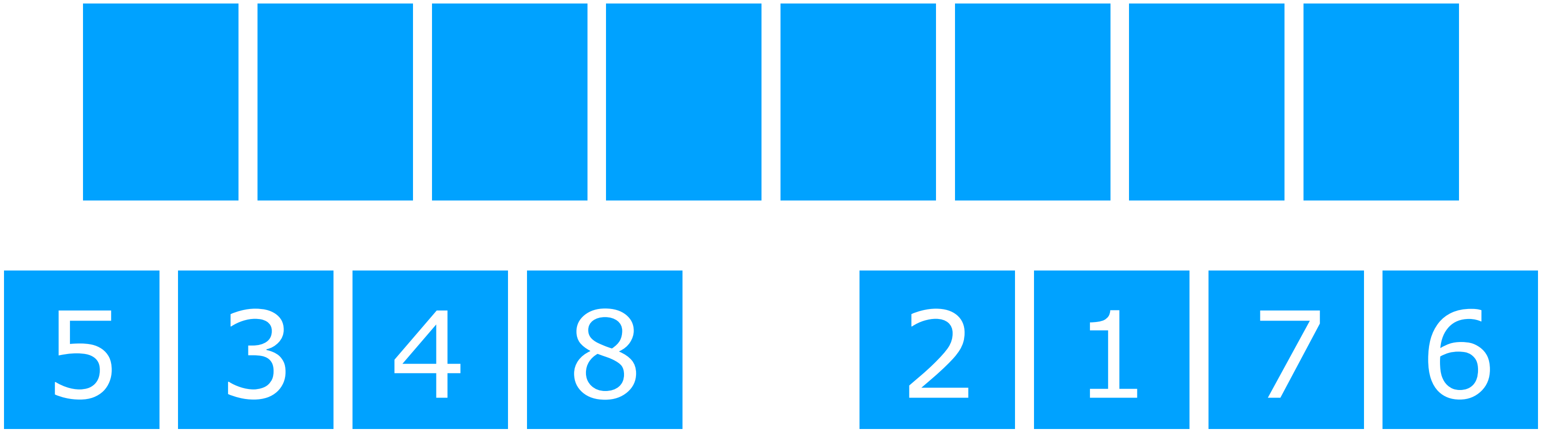
In the best case, I need to do approximately N^2 steps if my input size is N.

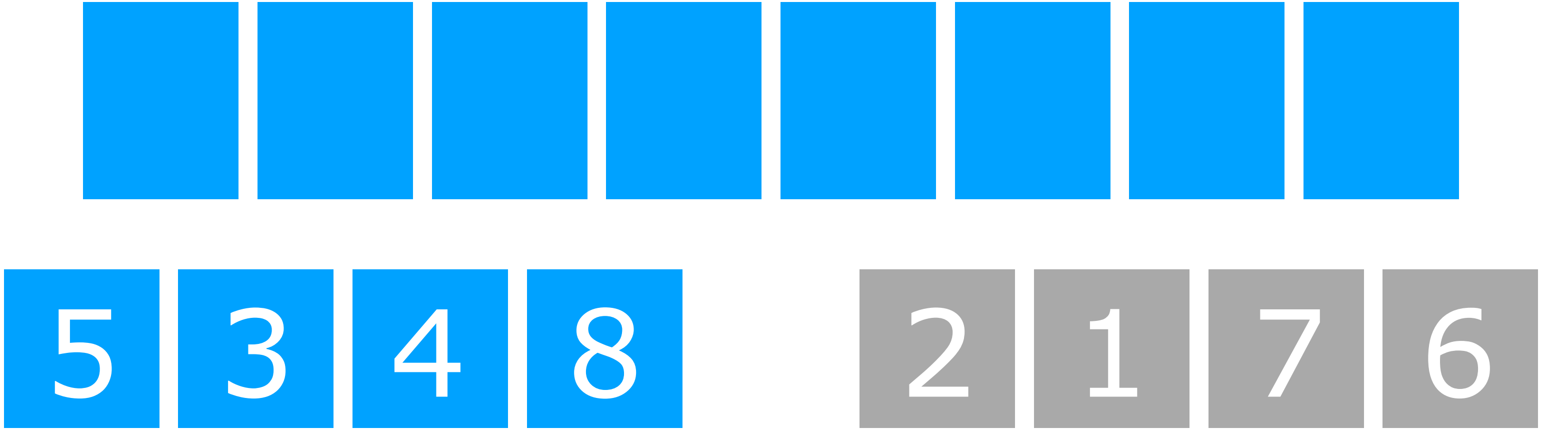
Merge Sort



5 3 4 8 2 1 7 6

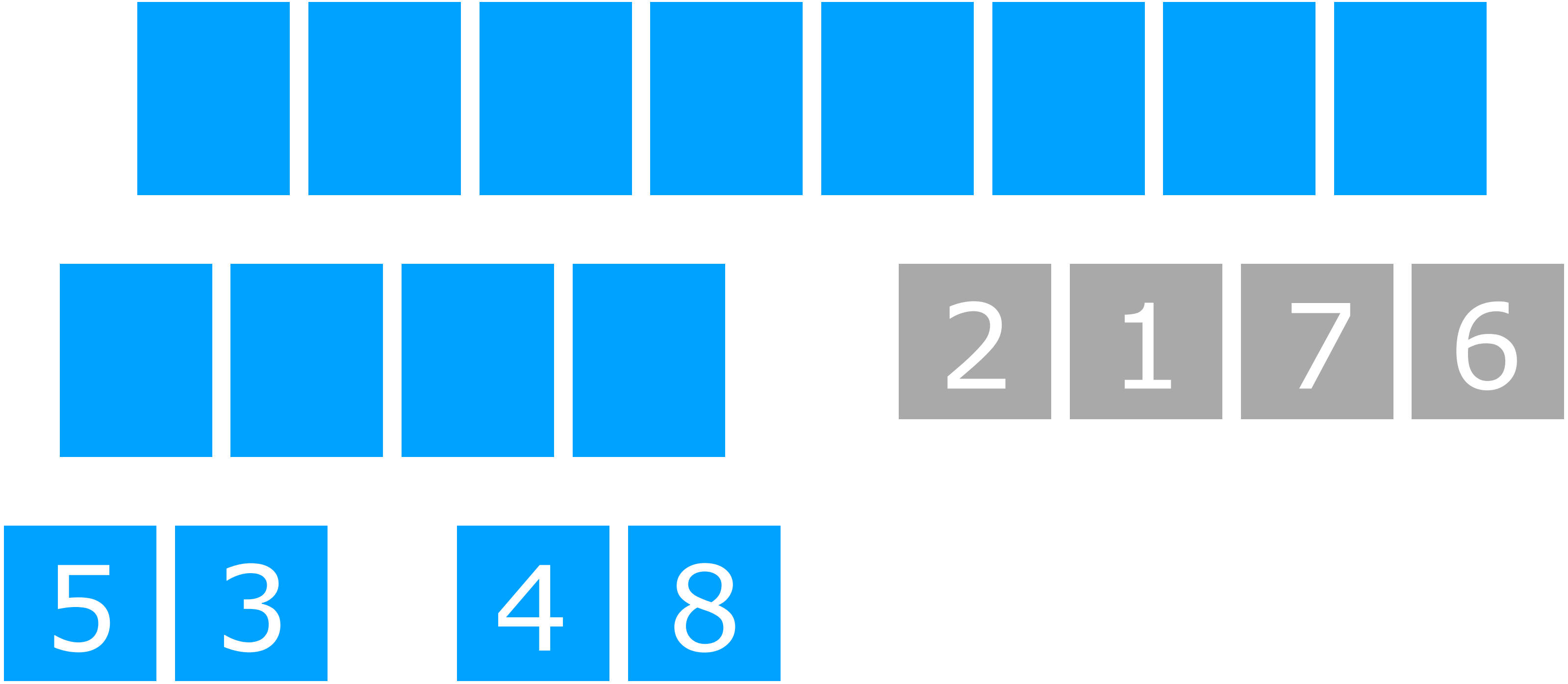
5	3	4	8	2	1	7	6

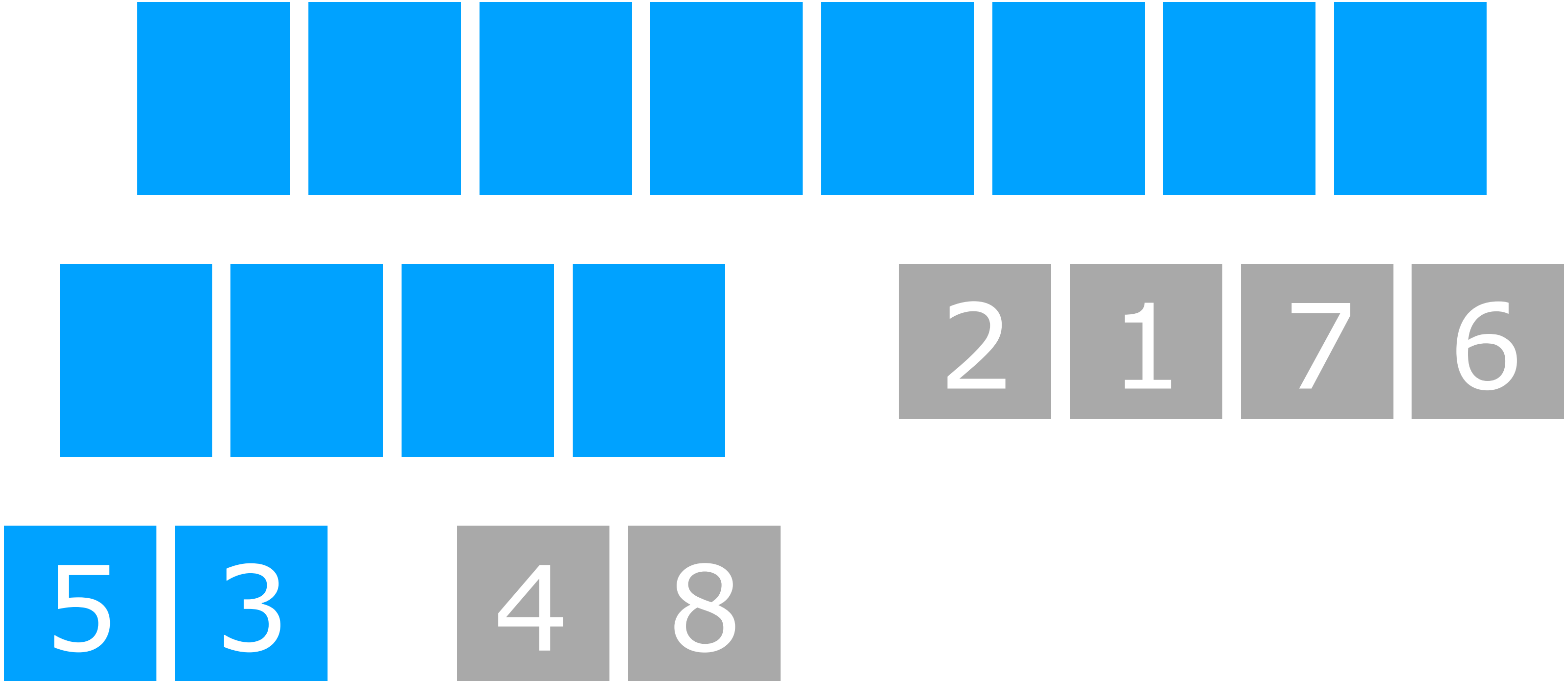

5 3 4 8
2 1 7 6

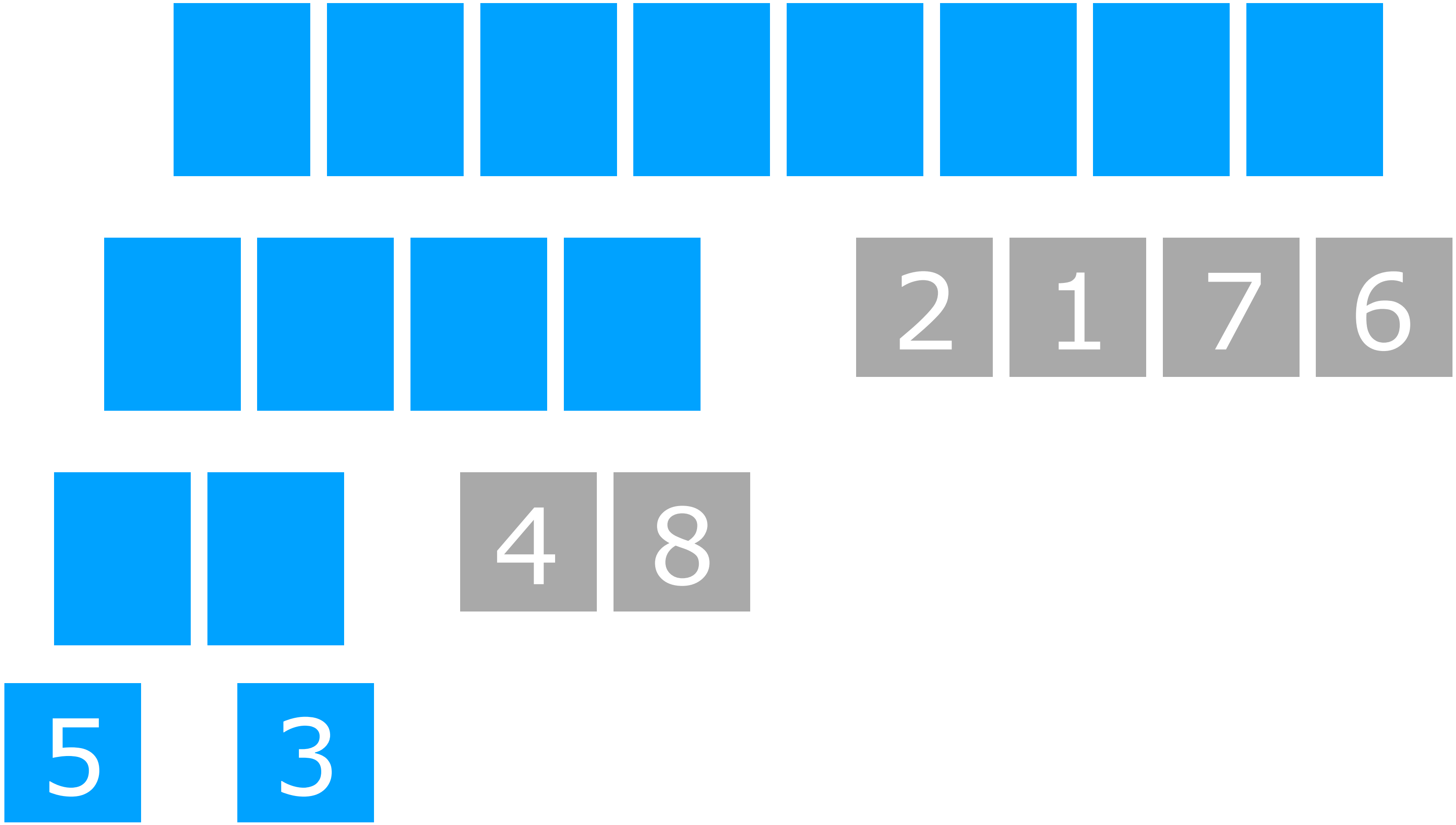


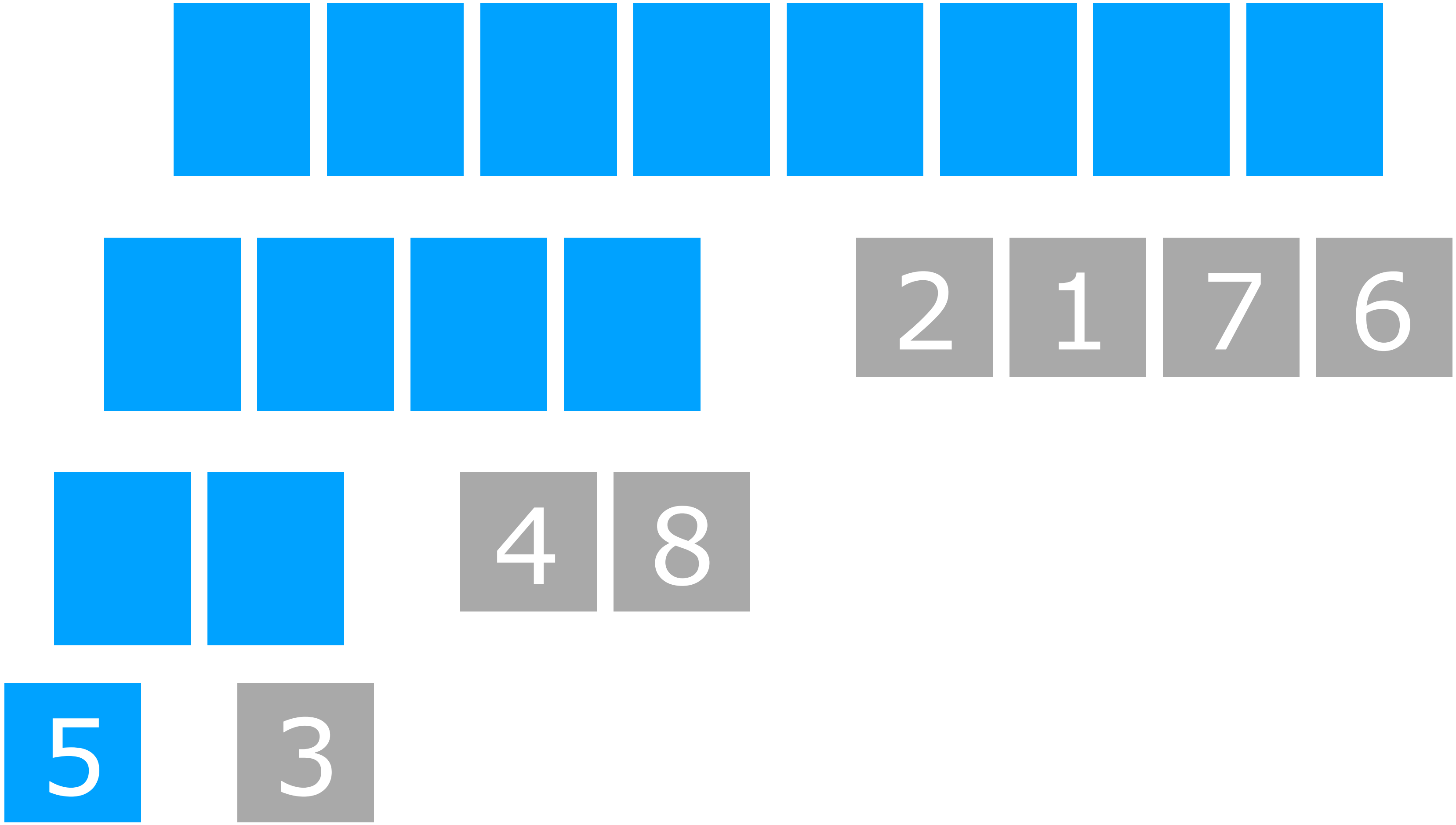
5 3 4 8

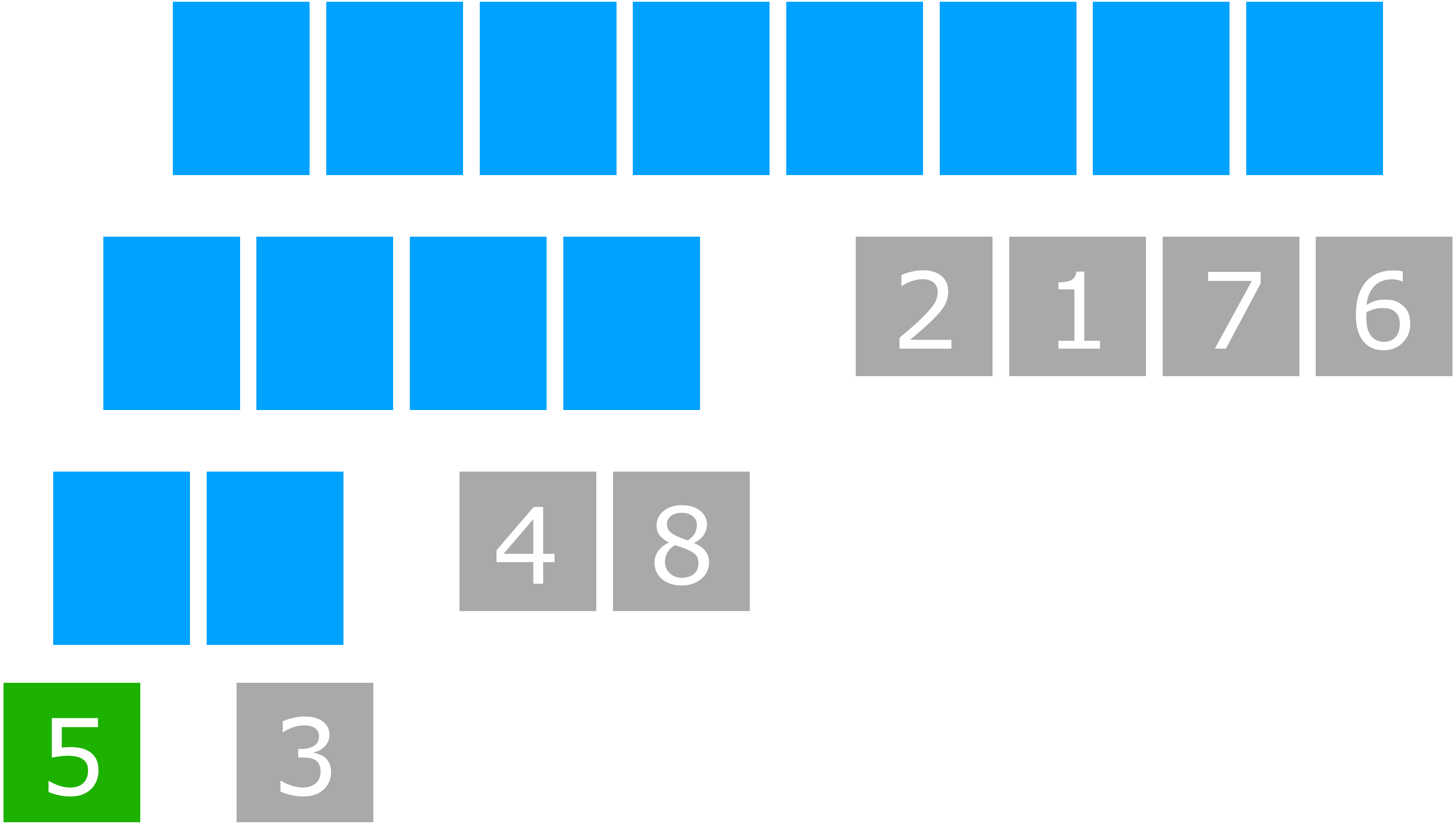
2 1 7 6

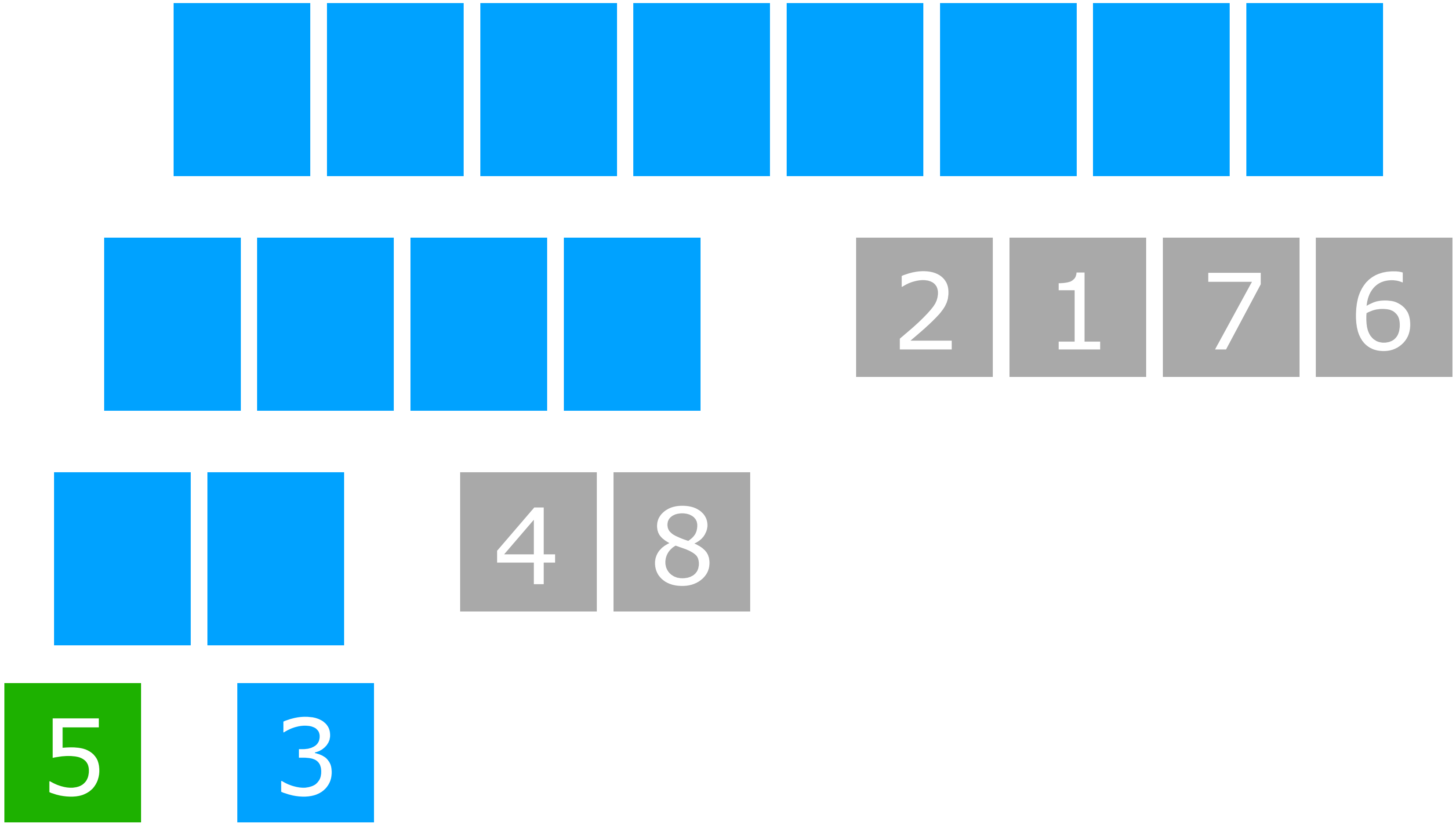

2
1
7
6
5
3
4
8

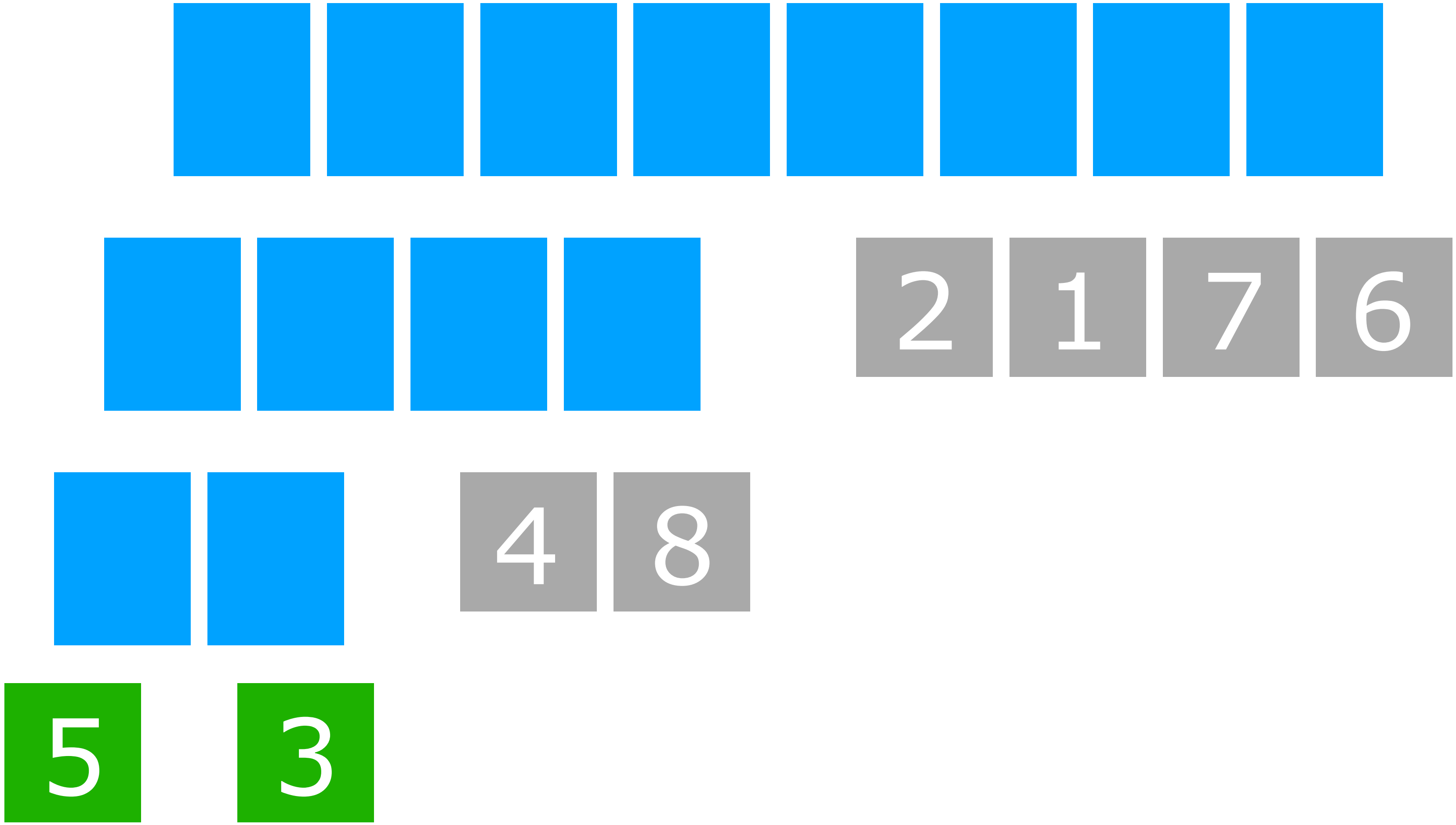

2
1
7
6
5
3
4
8

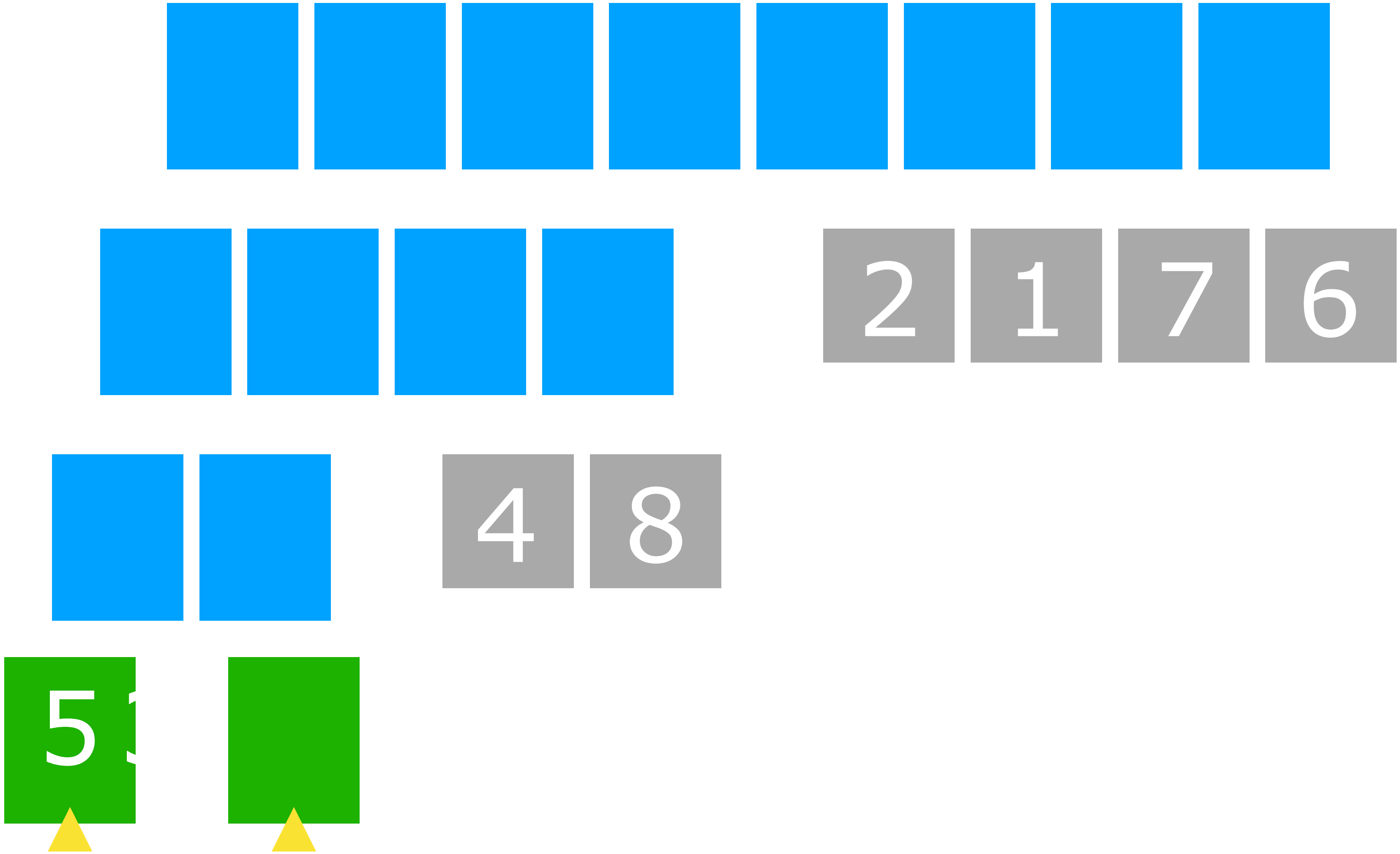

2
1
7
6
4
8
5
3


2
1
7
6
4
8
5
3

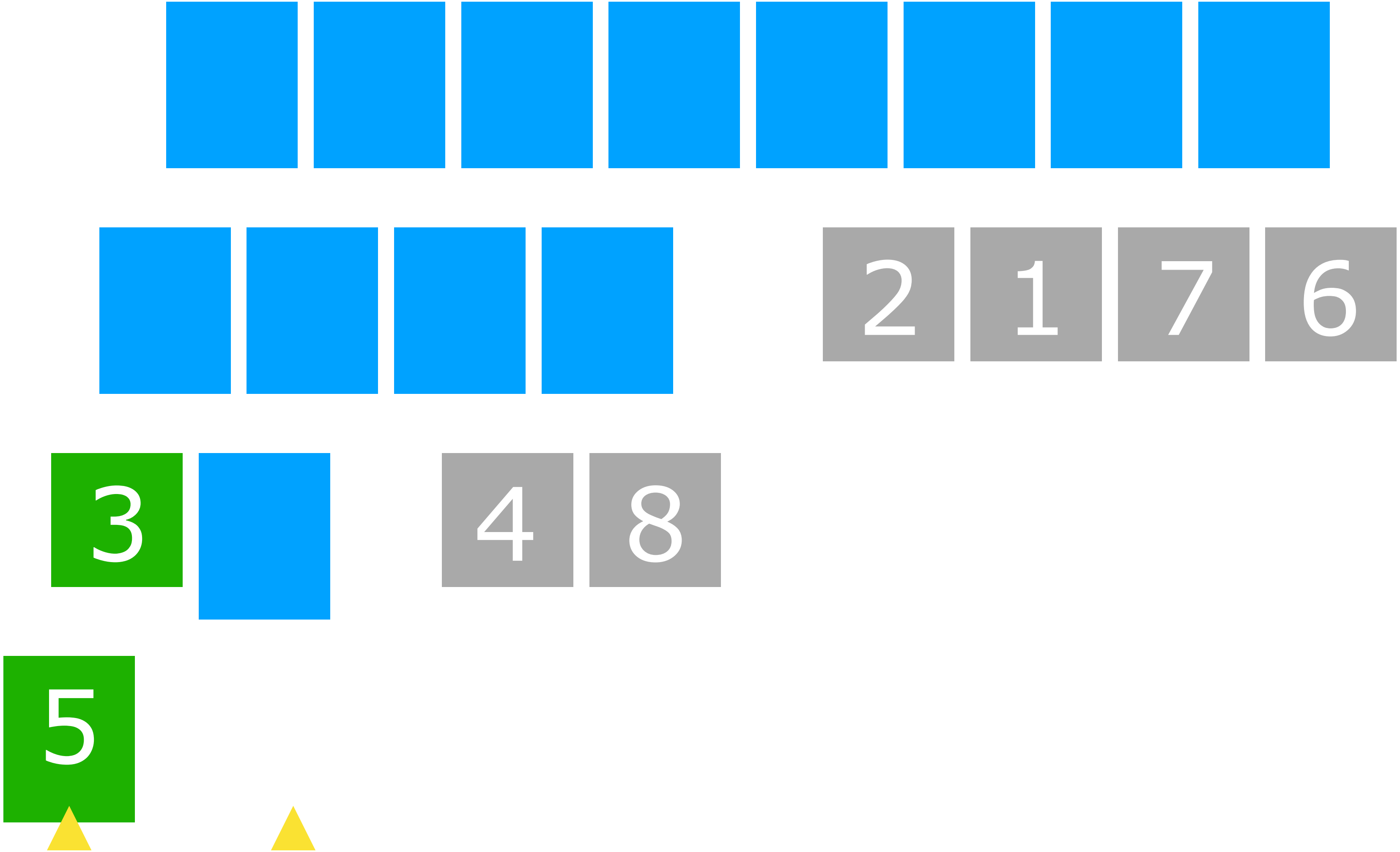

2
1
7
6
4
8
5
3

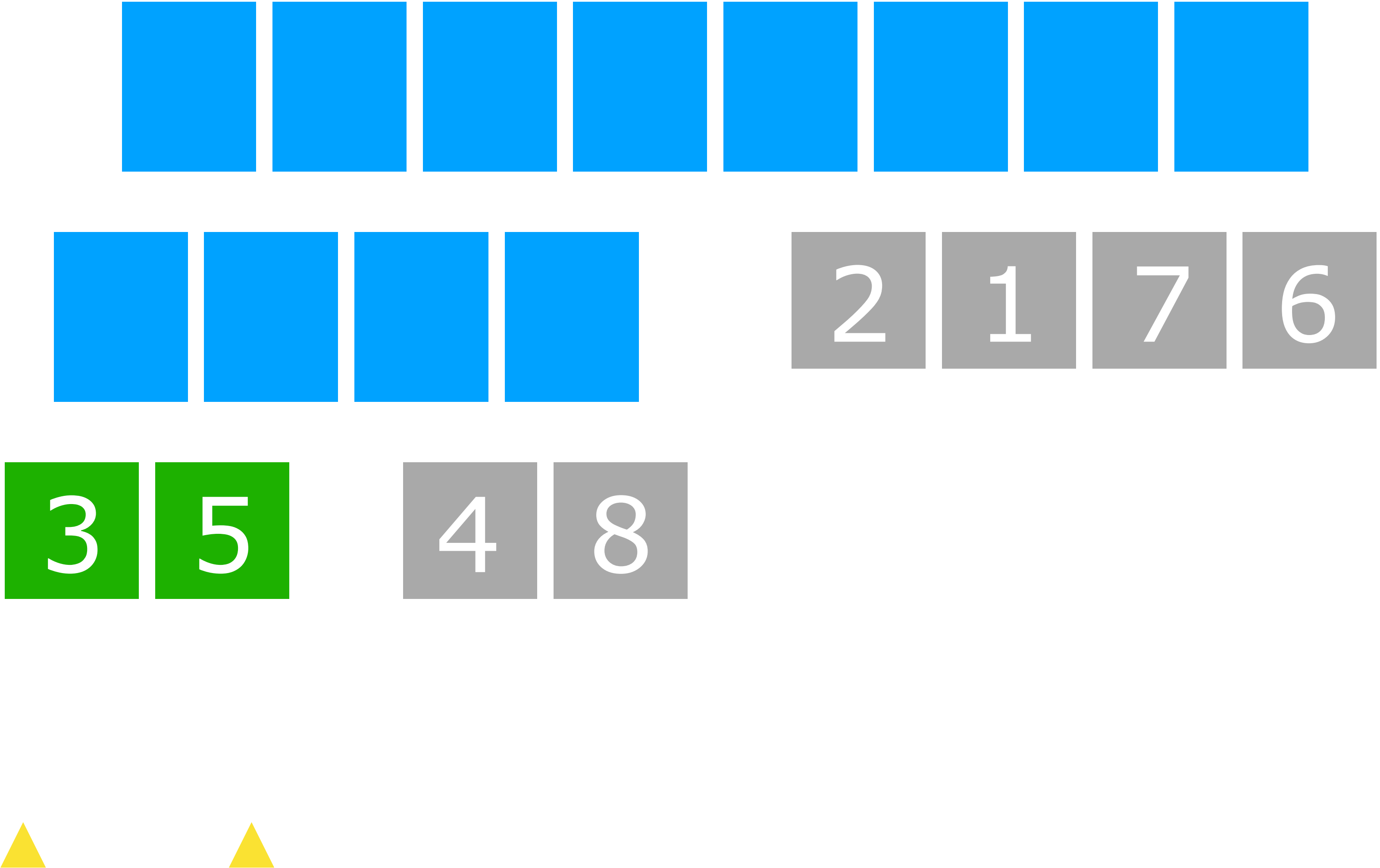

2
1
7
6
4
8
5
3

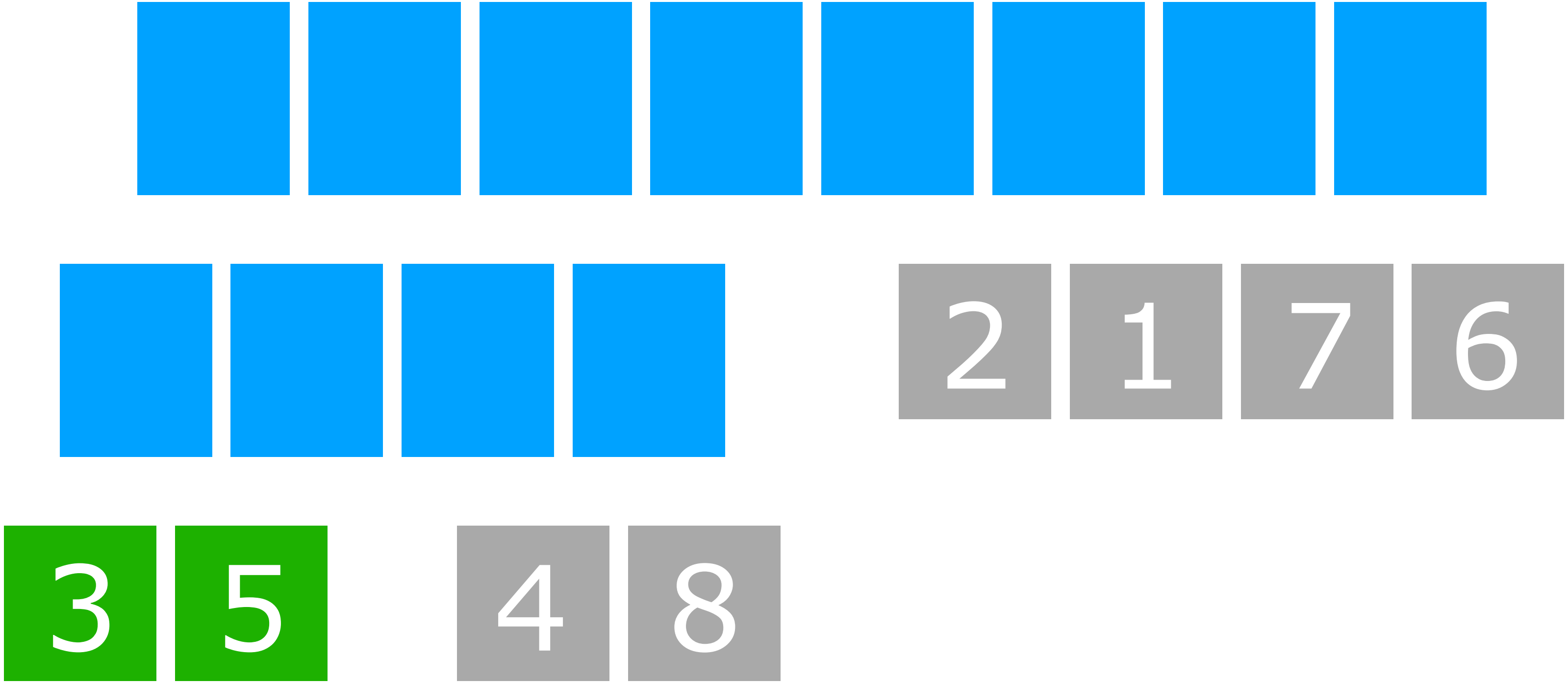

2
1
7
6
4
8
5
3

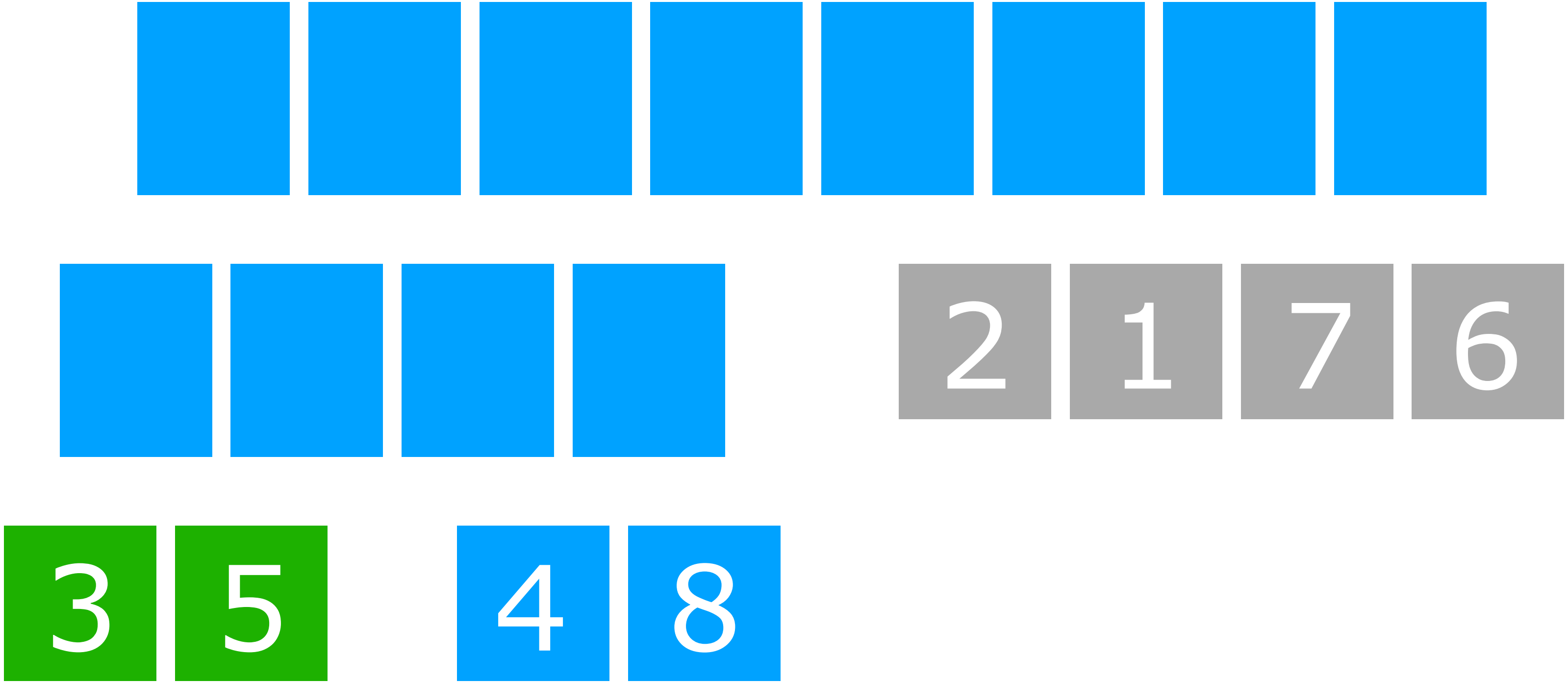


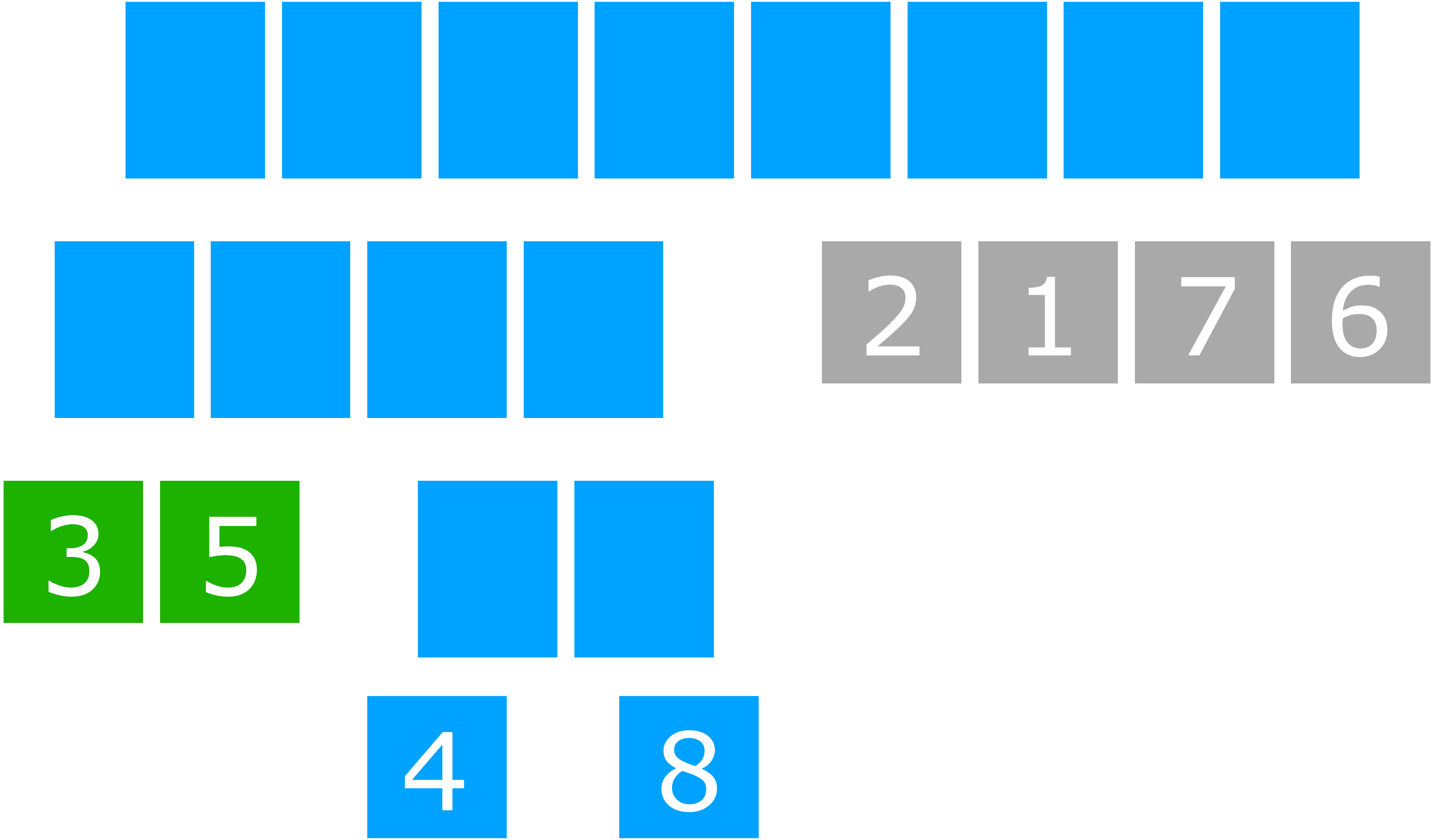
2
1
7
6
4
8
5

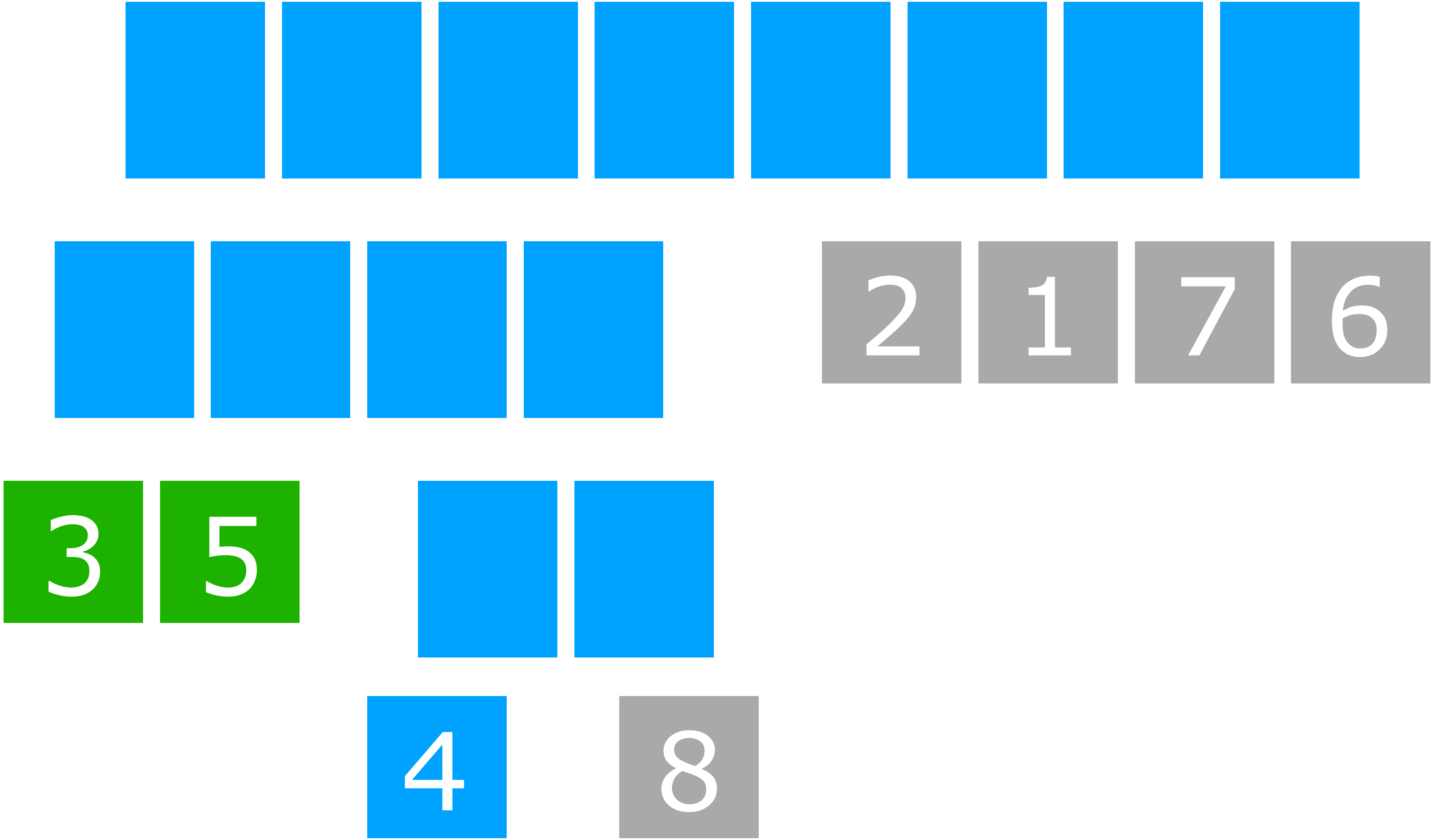

2
1
7
6
3
4
8
5

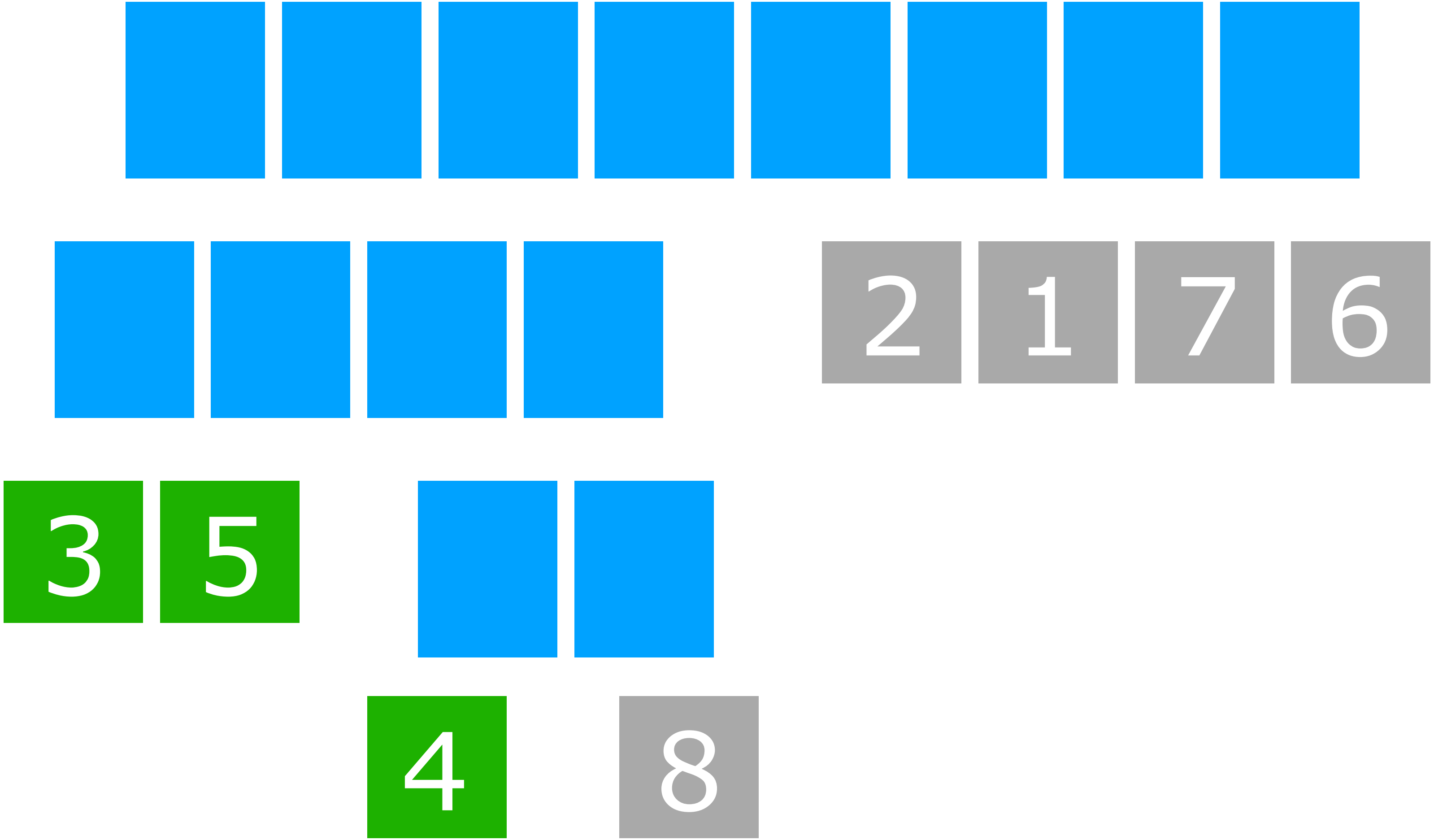

2
1
7
6
3
5
4
8

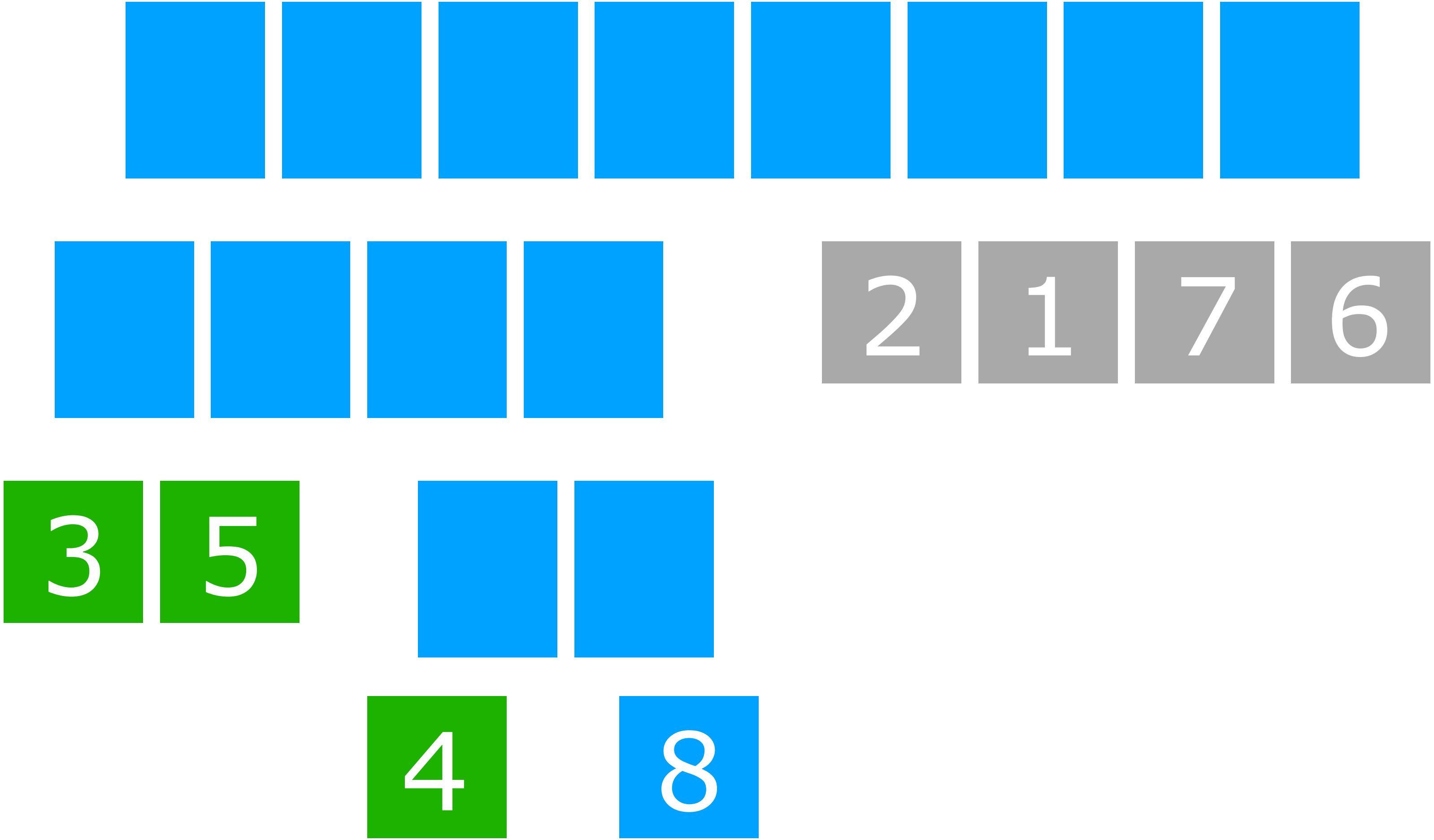

2 1 7 6
3 5
4 8

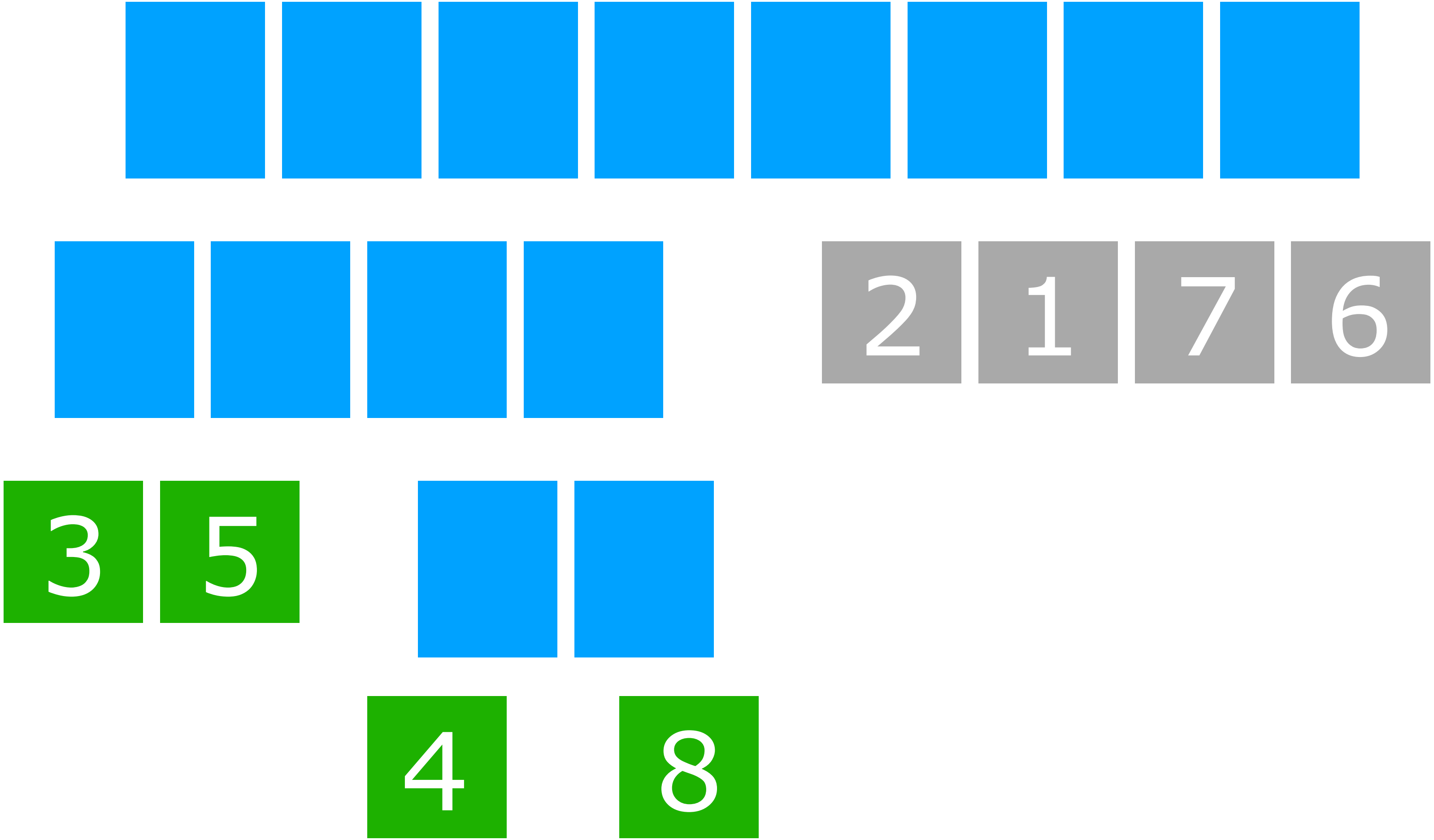

2
1
7
6
3
5
4
8

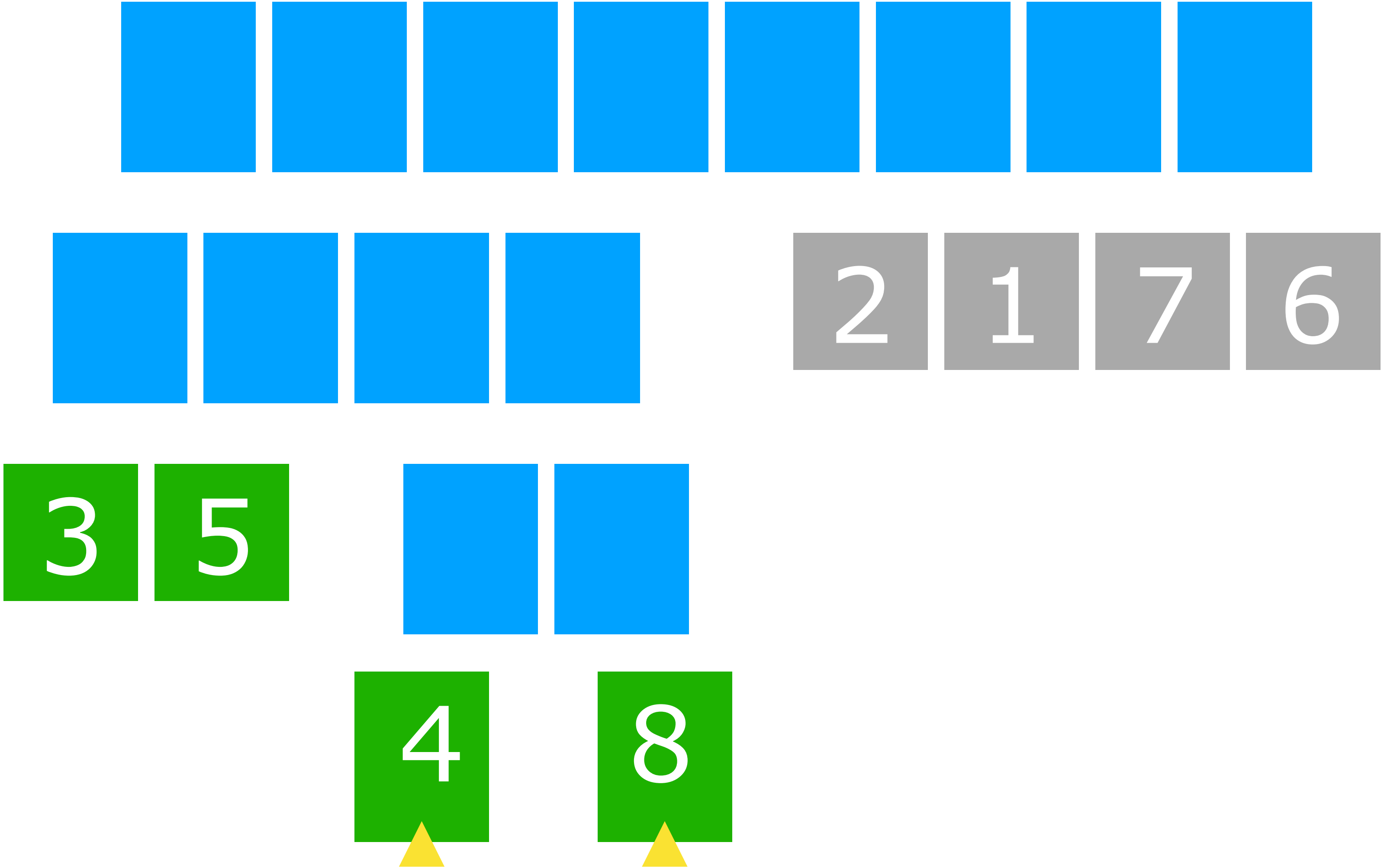

2
1
7
6
3
5
4
8

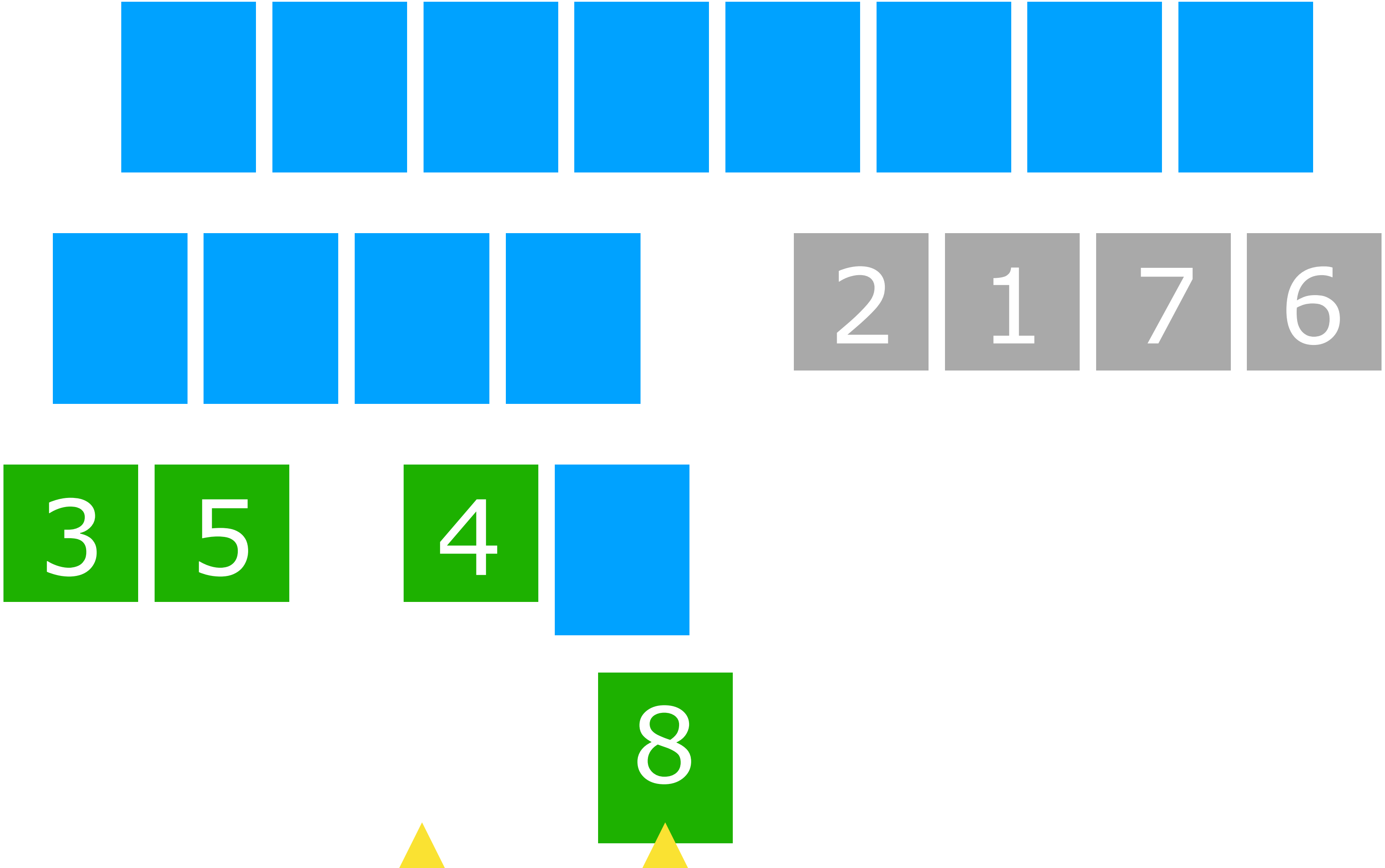

2
1
7
6
3
5
4
8

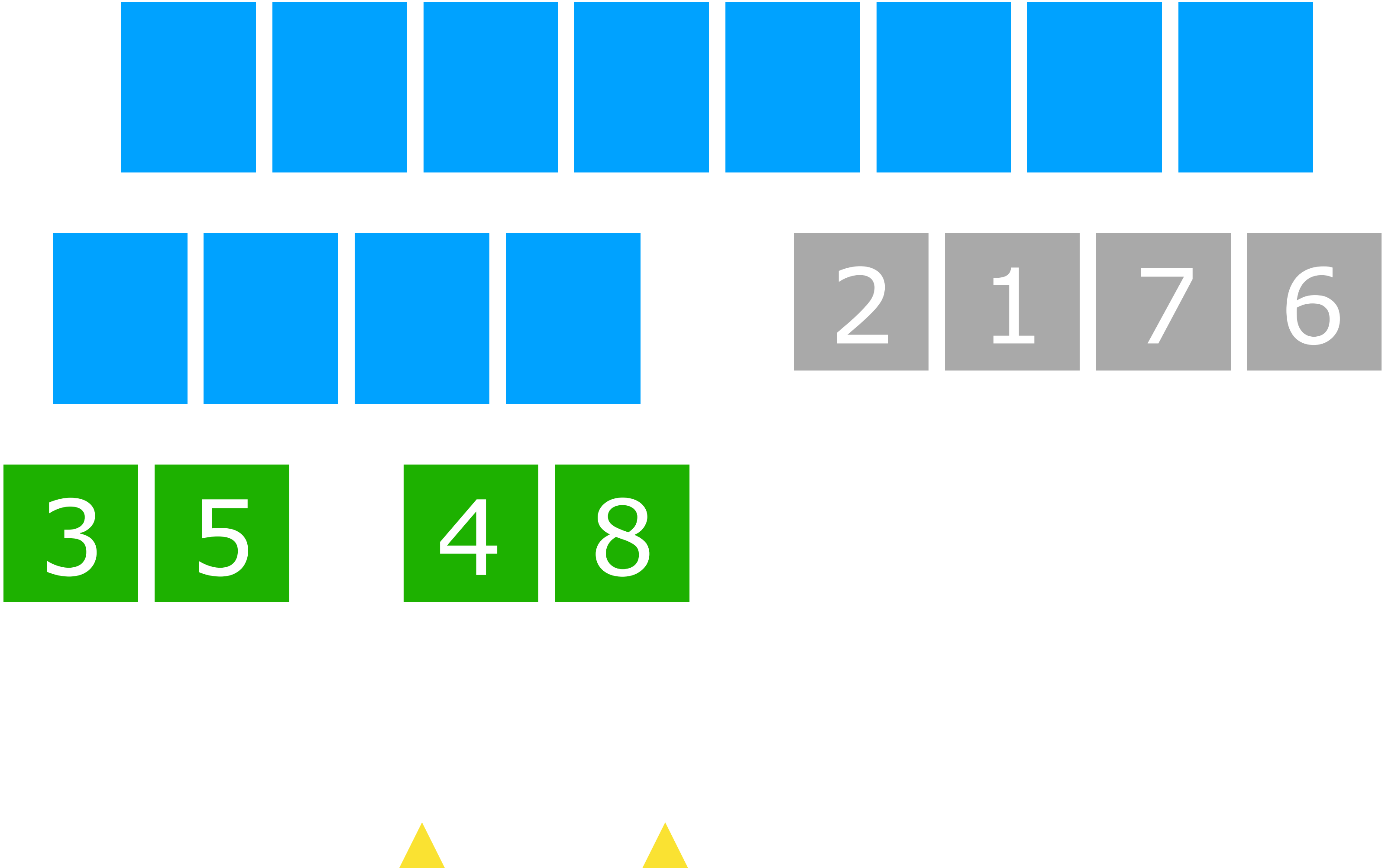

2
1
7
6
3
5
4
8

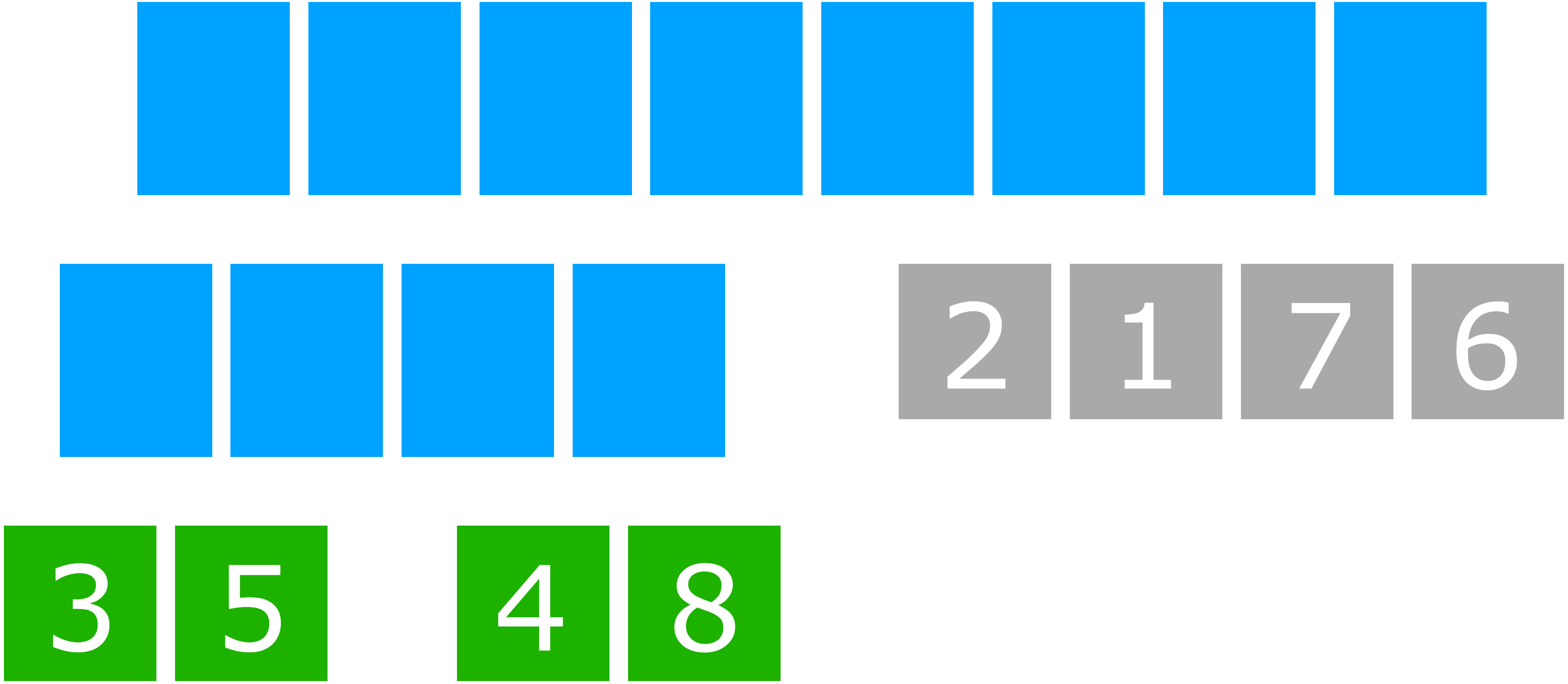

2
1
7
6
3
5
4
8

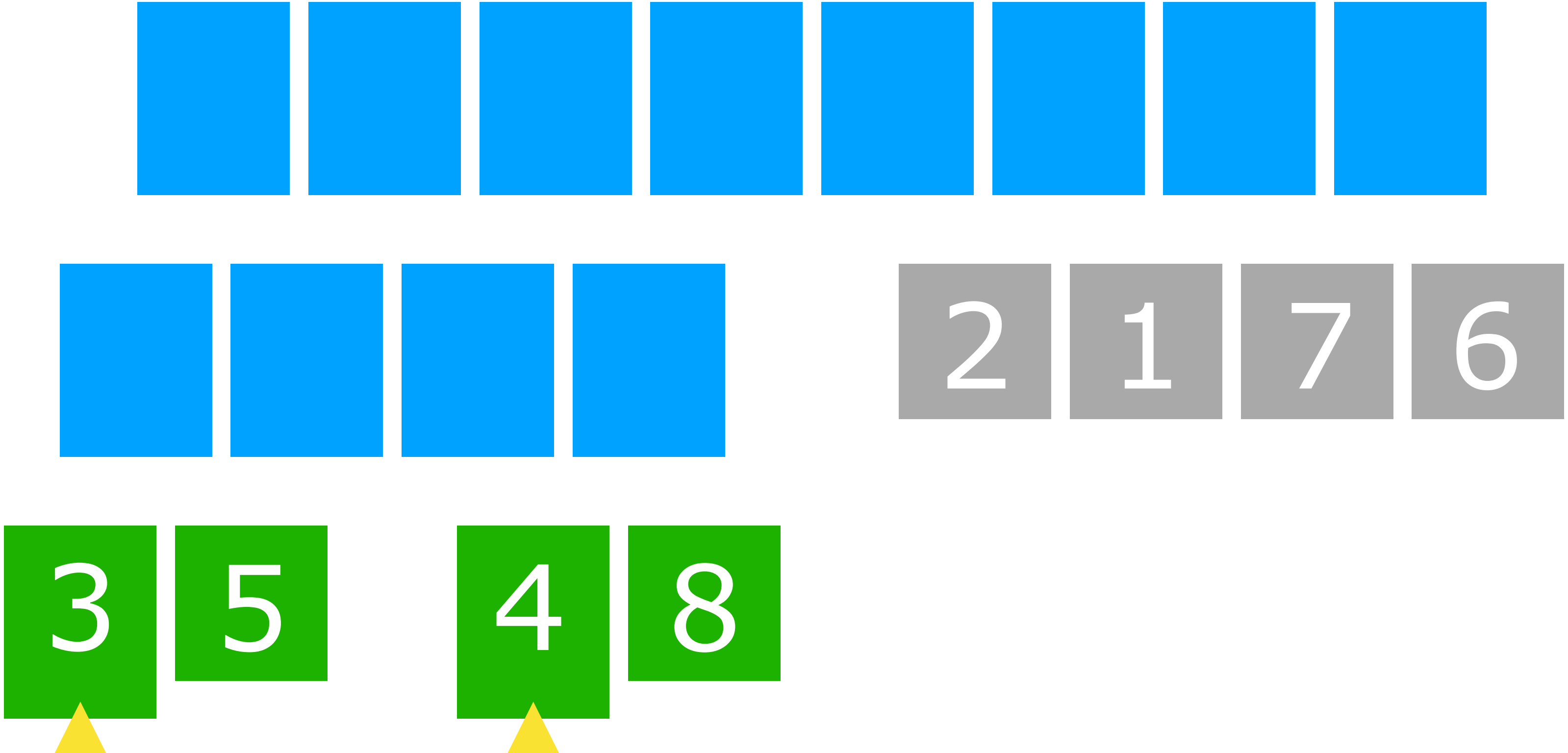

2
1
7
6
3
5
4
8

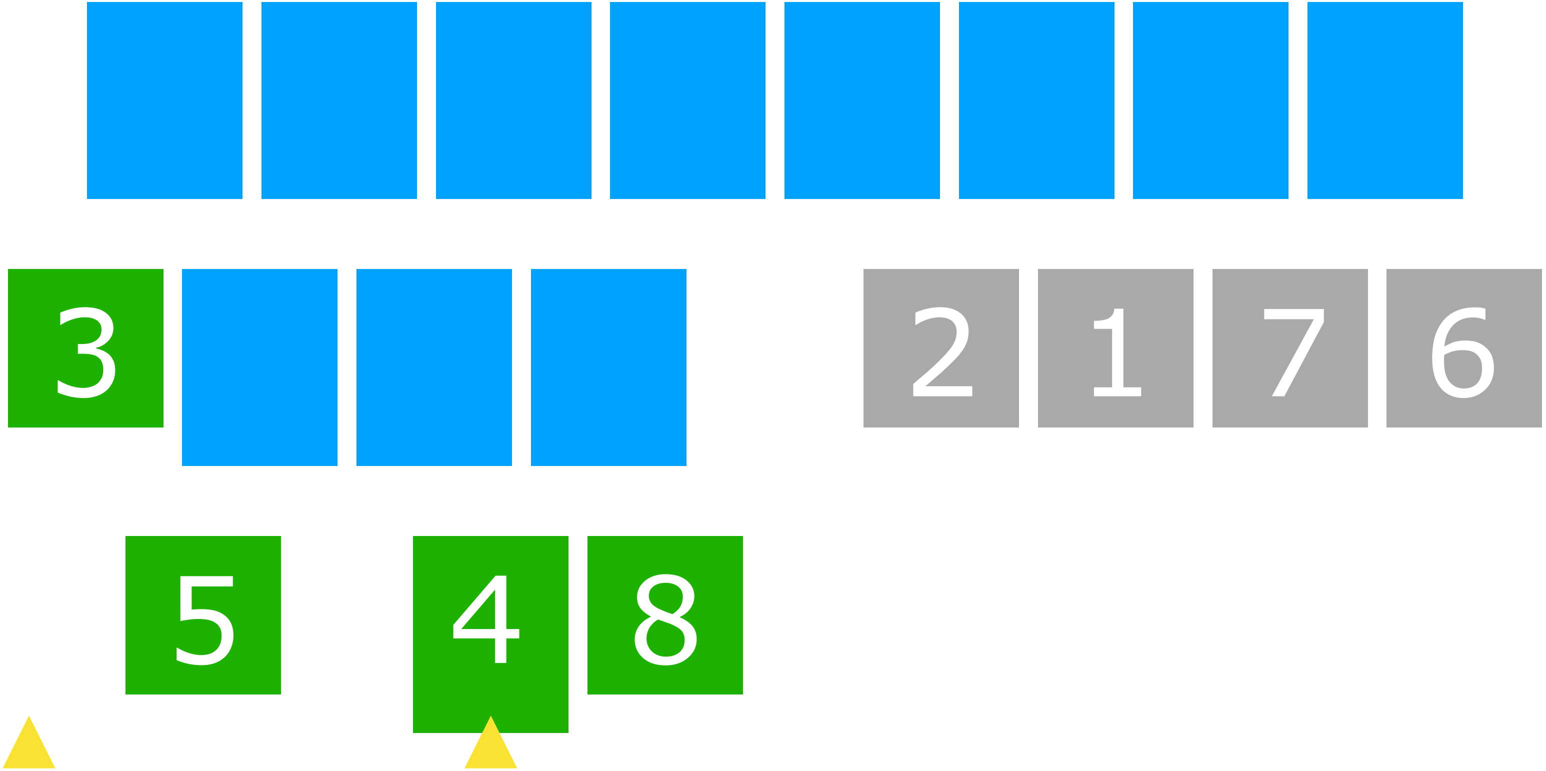

2
1
7
6
3
5
4
8

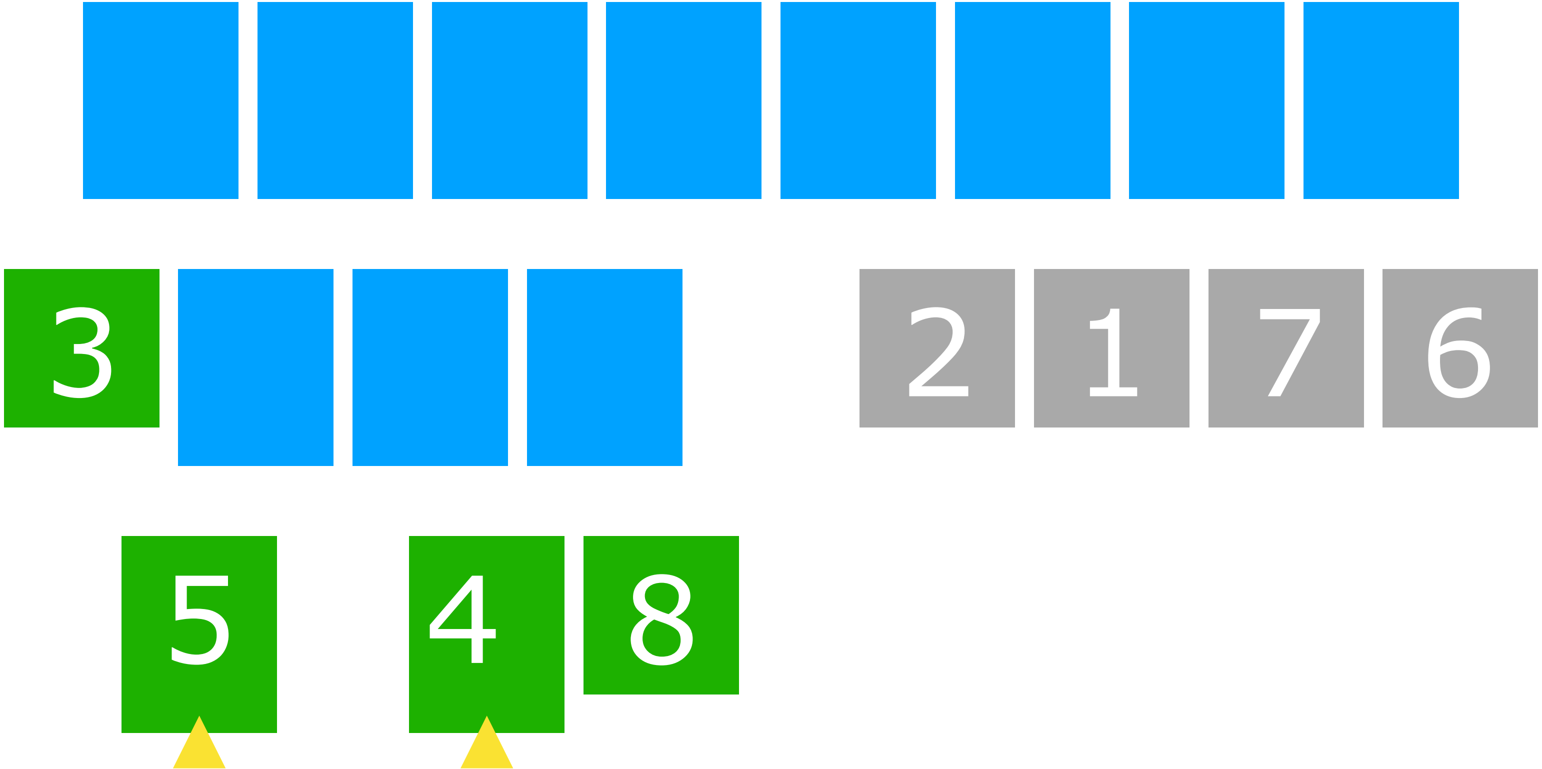

2
1
7
6
3
5
4
8

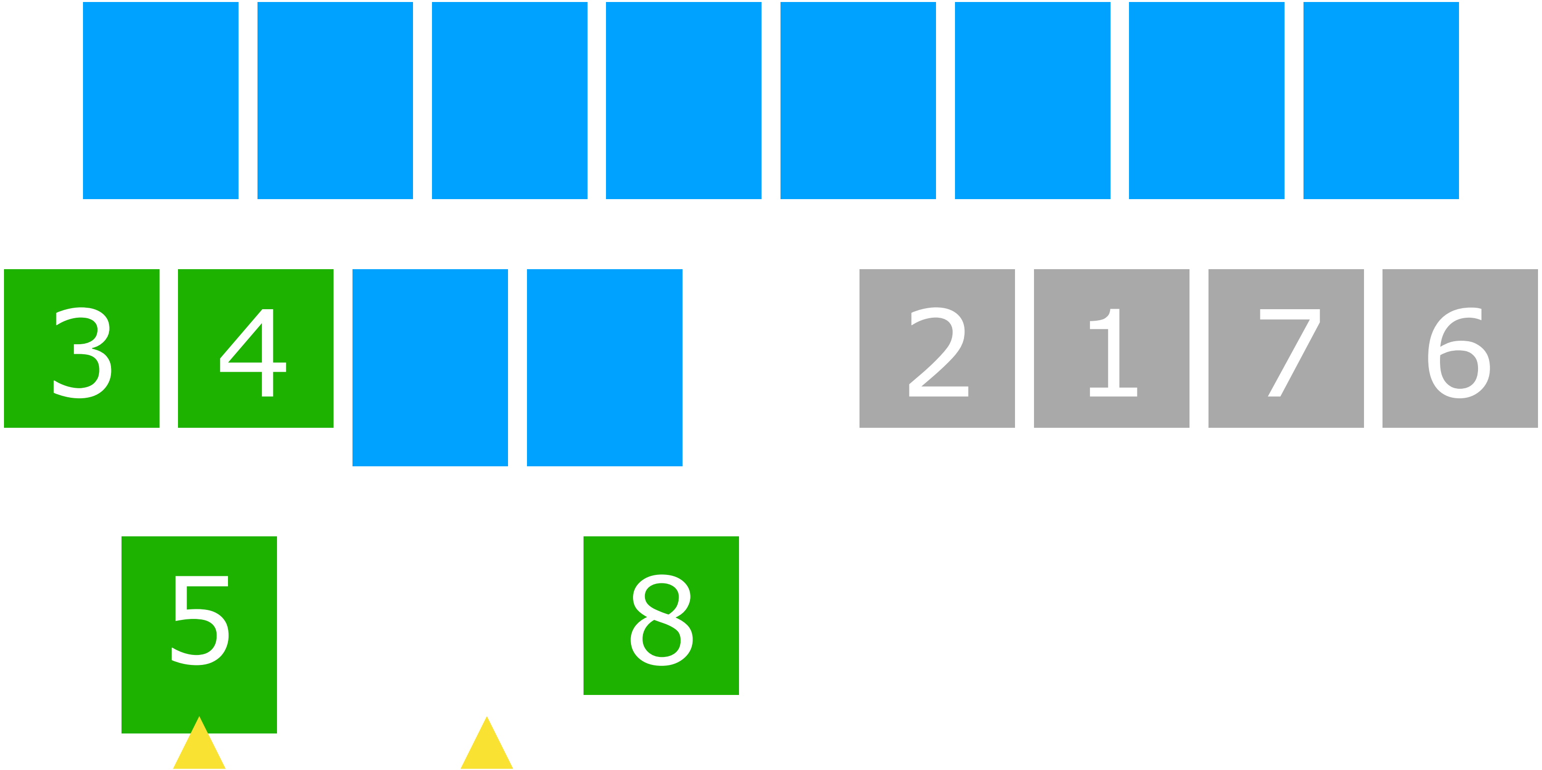

2
1
7
6
3
5
4
8

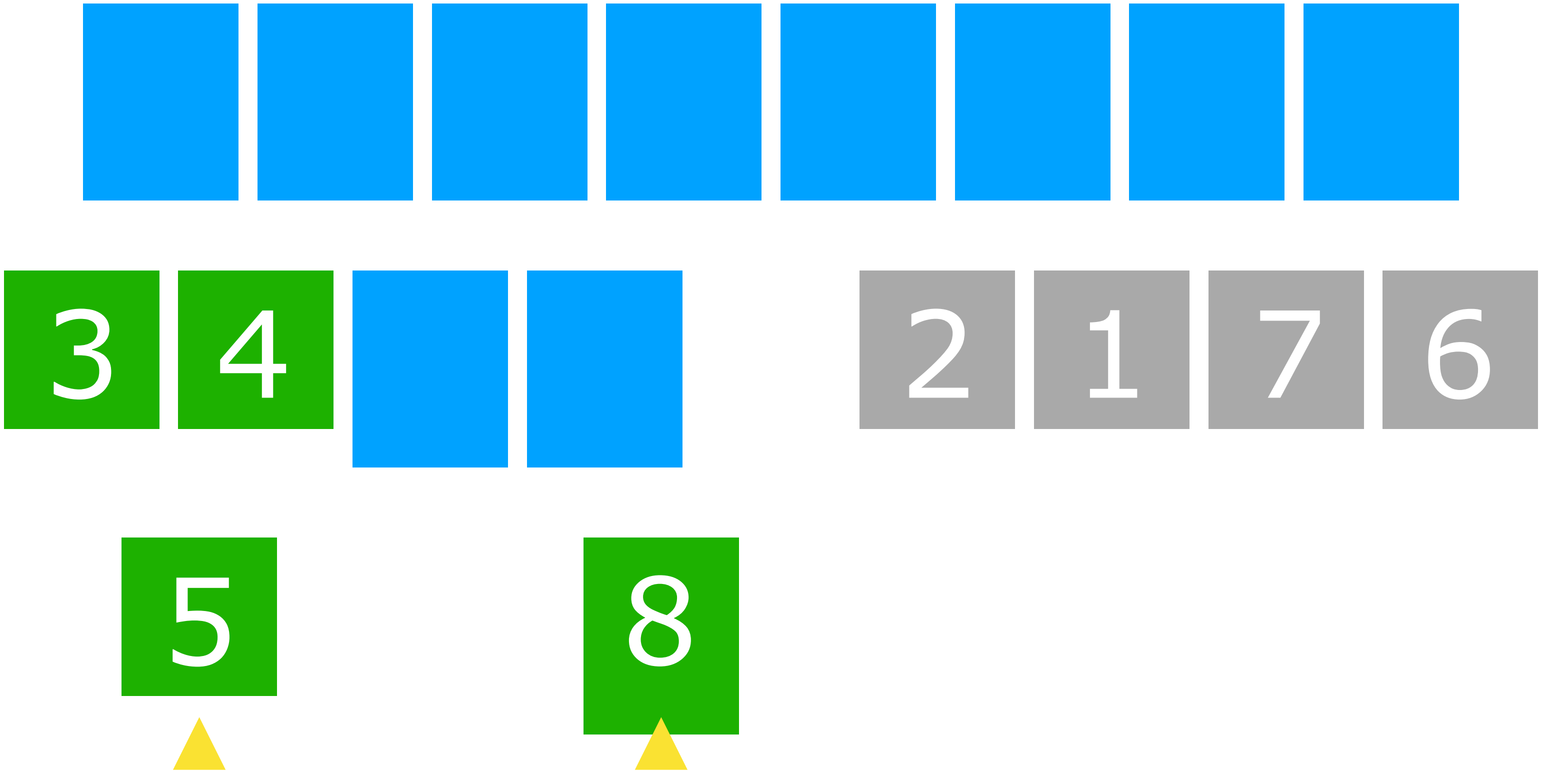

2
1
7
6
3
5
4
8

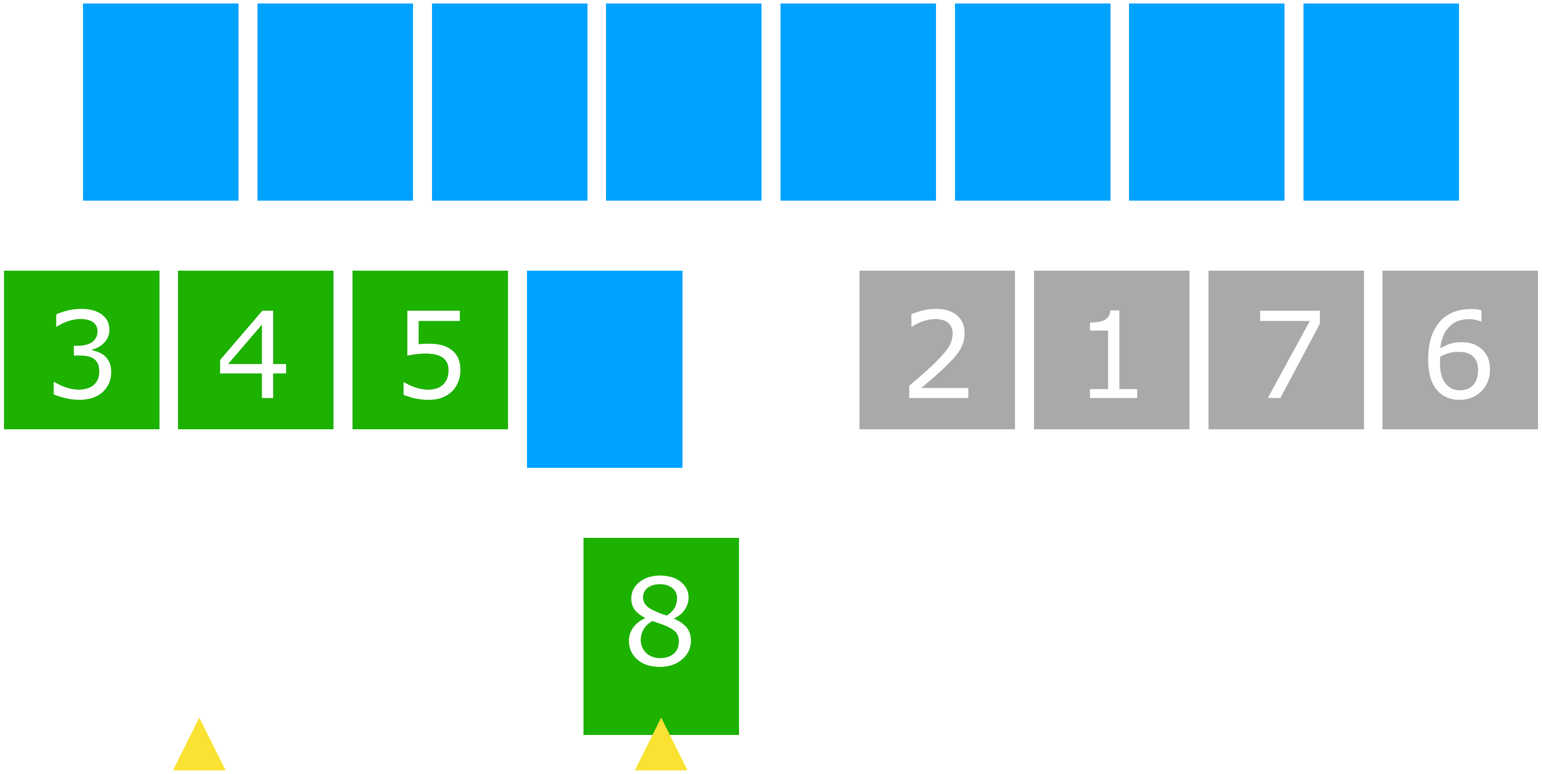

2
1
7
6
3
5
4
8

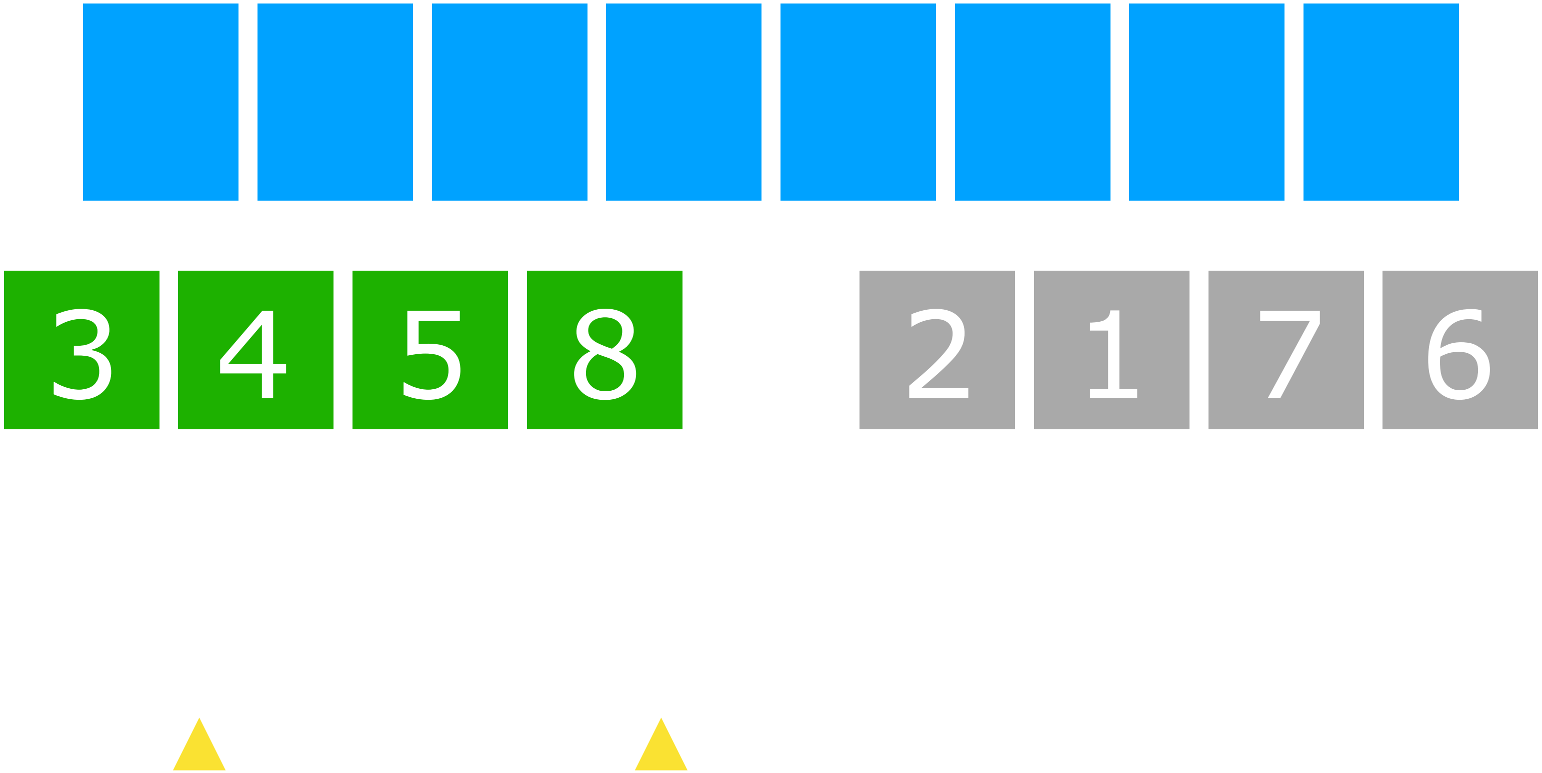

3
2
1
7
6
5
4
8


3
2
1
7
6
5
4
8


3
4
2
1
7
6
5
8

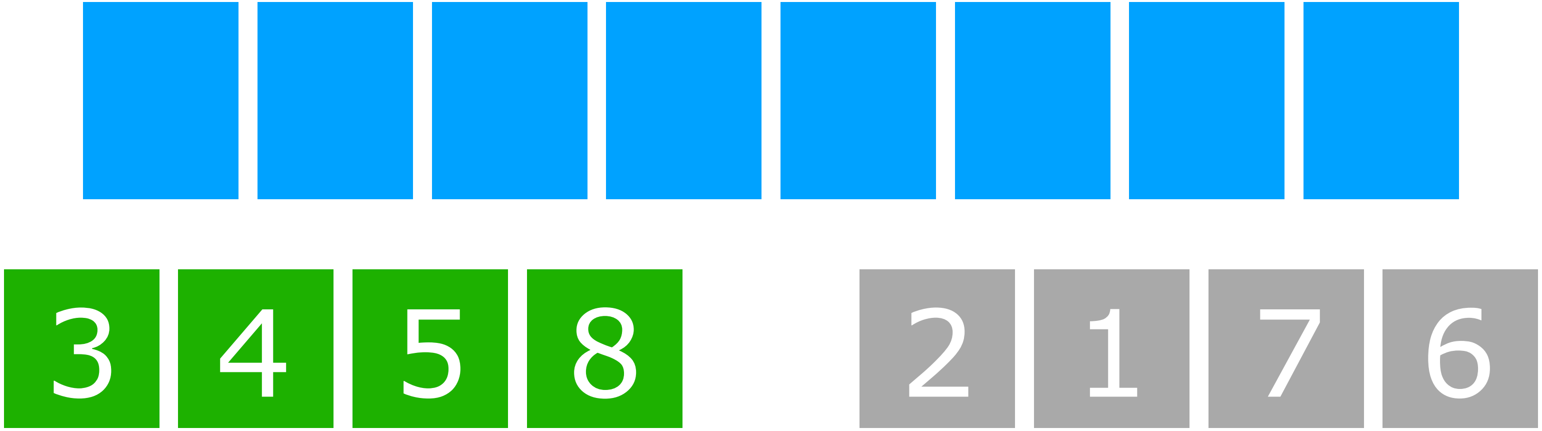

3
4
2
1
7
6
5
8


3
4
5
2
1
7
6
8



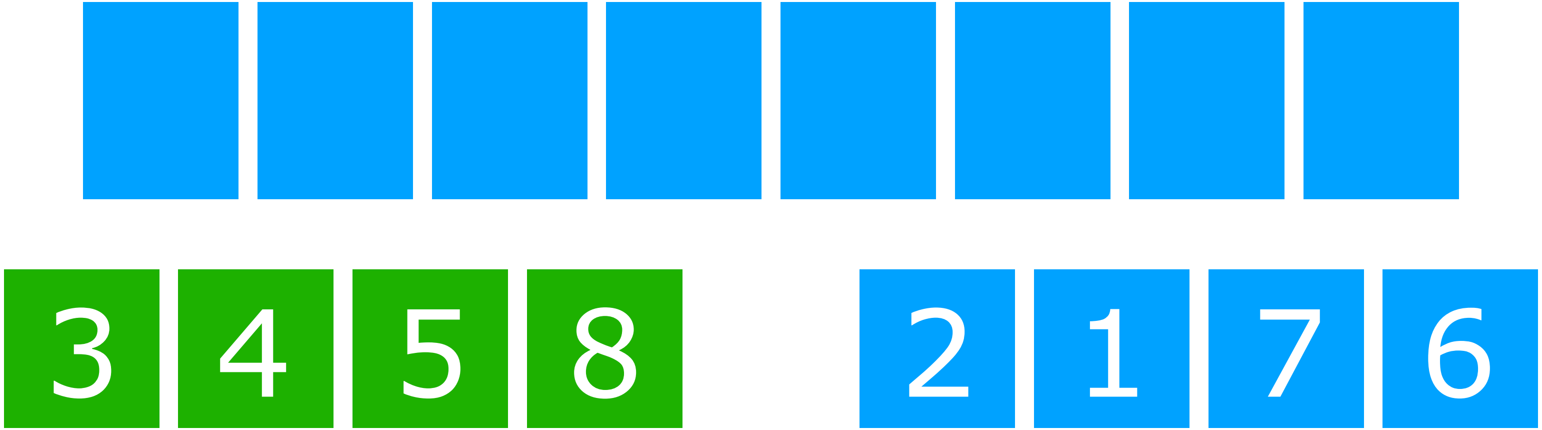
3 4 5 8

2 1 7 6



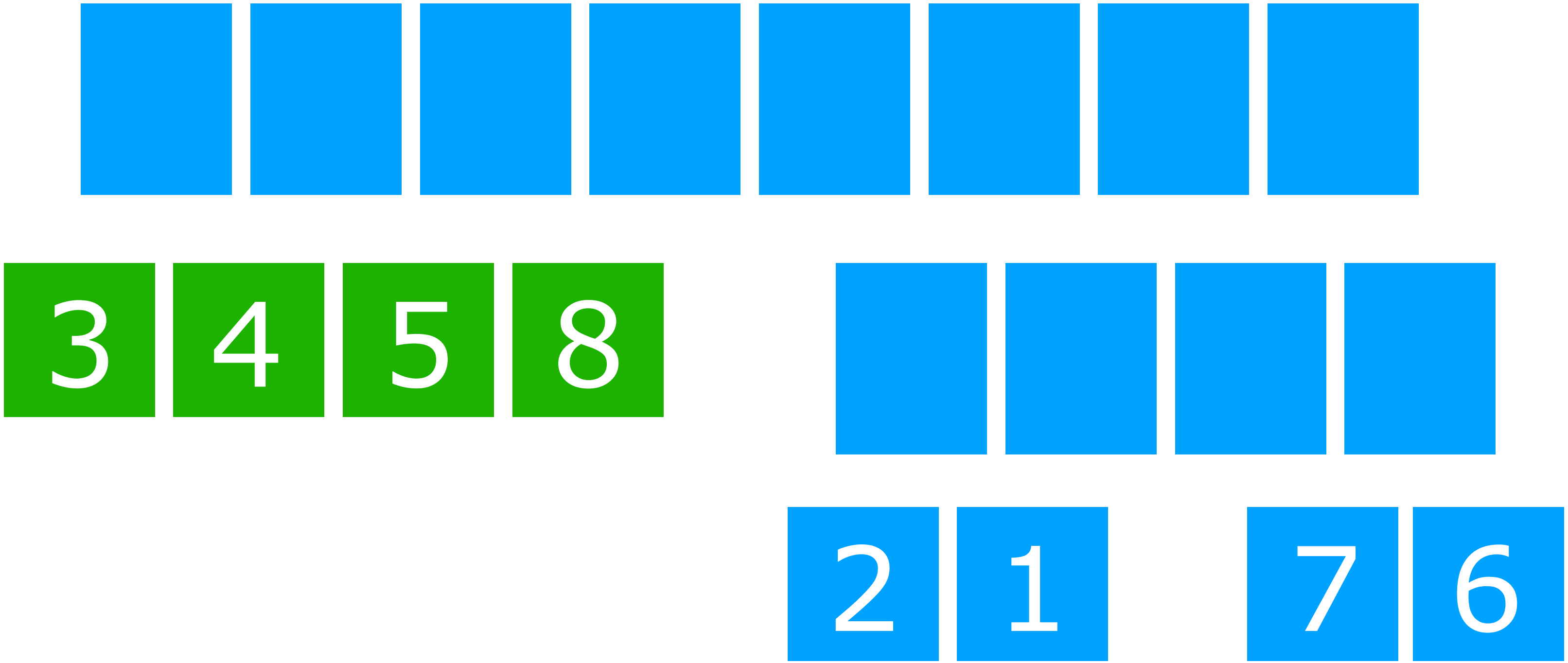
3 4 5 8

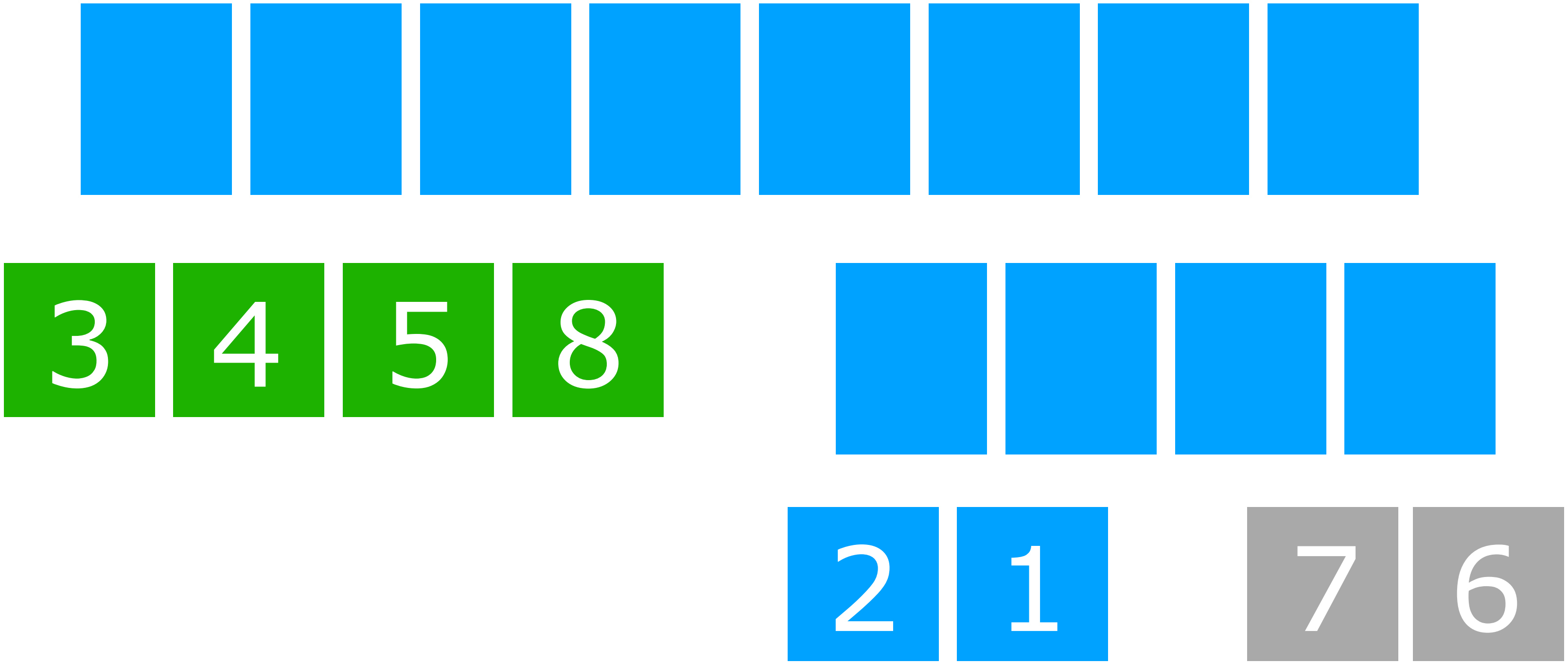
2 1 7 6

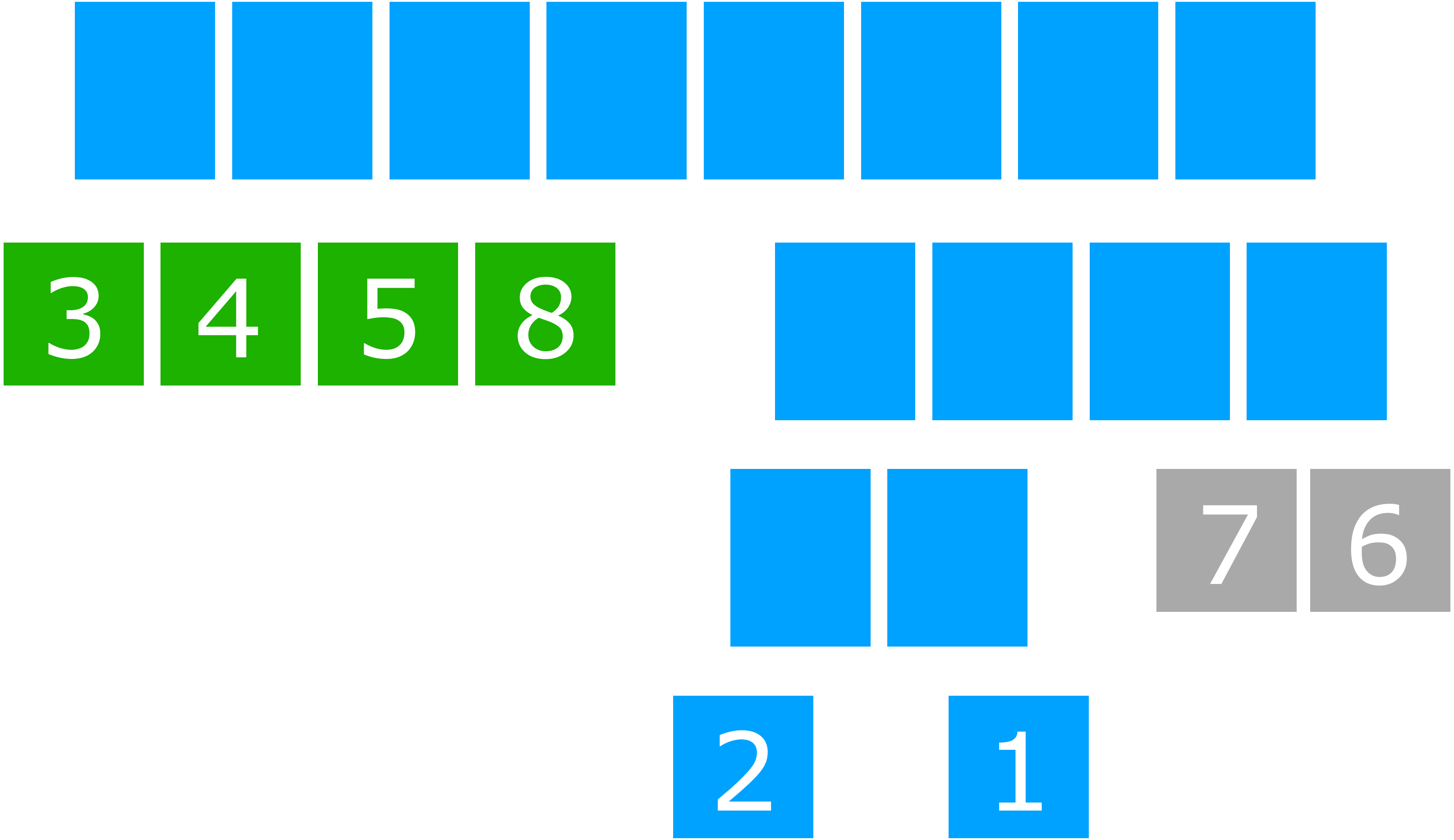


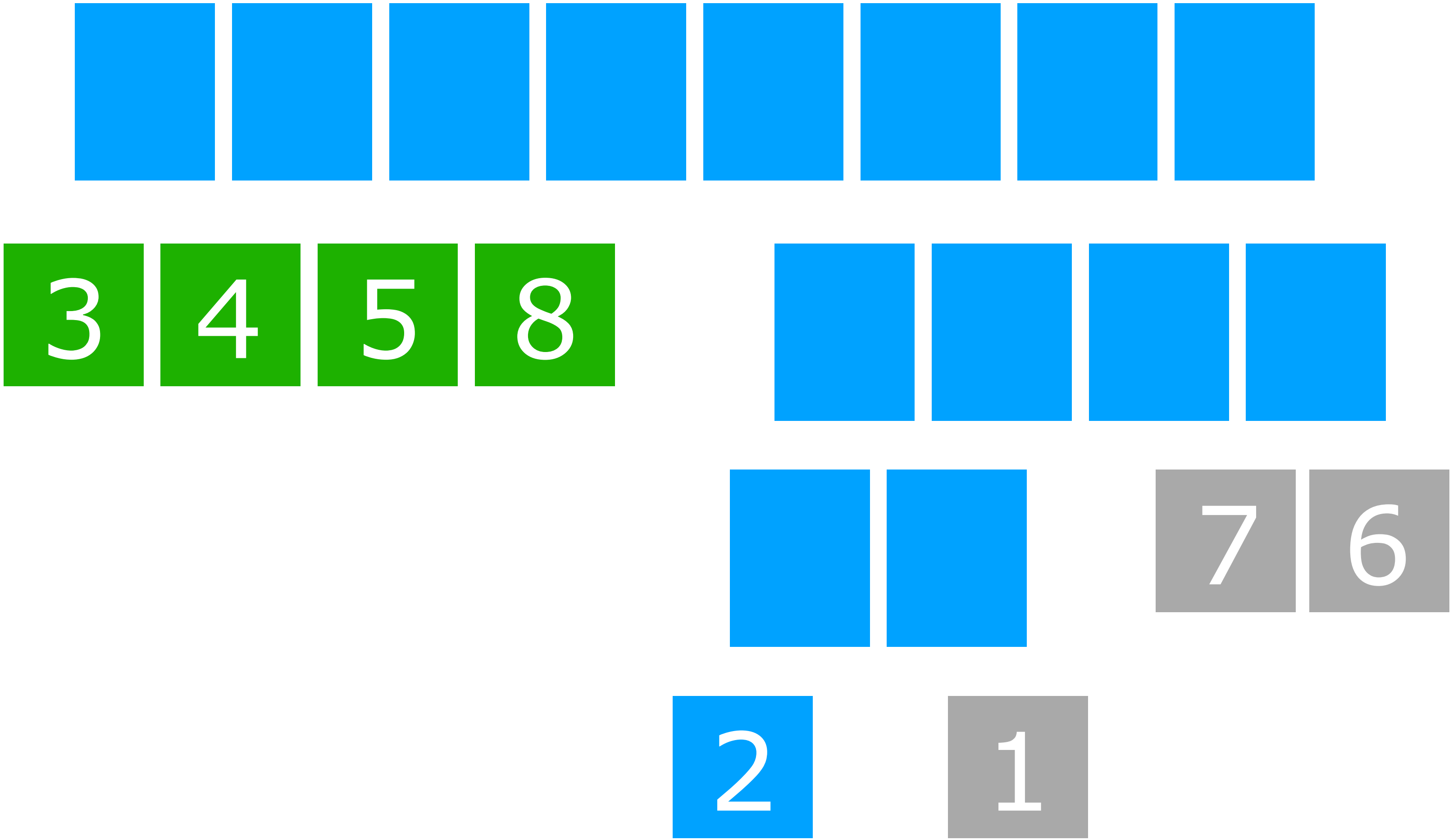
3 4 5 8

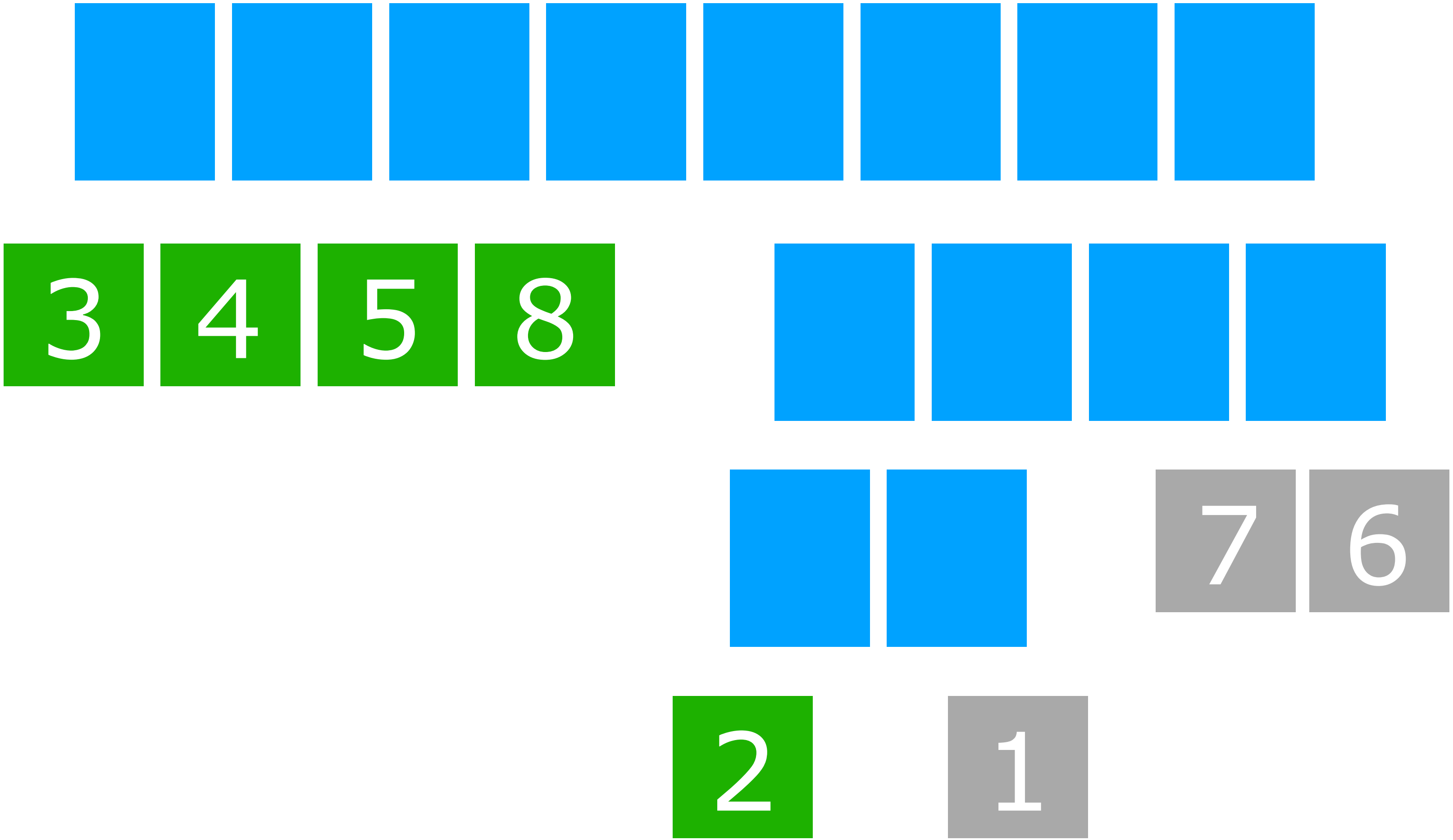
2 1 7 6

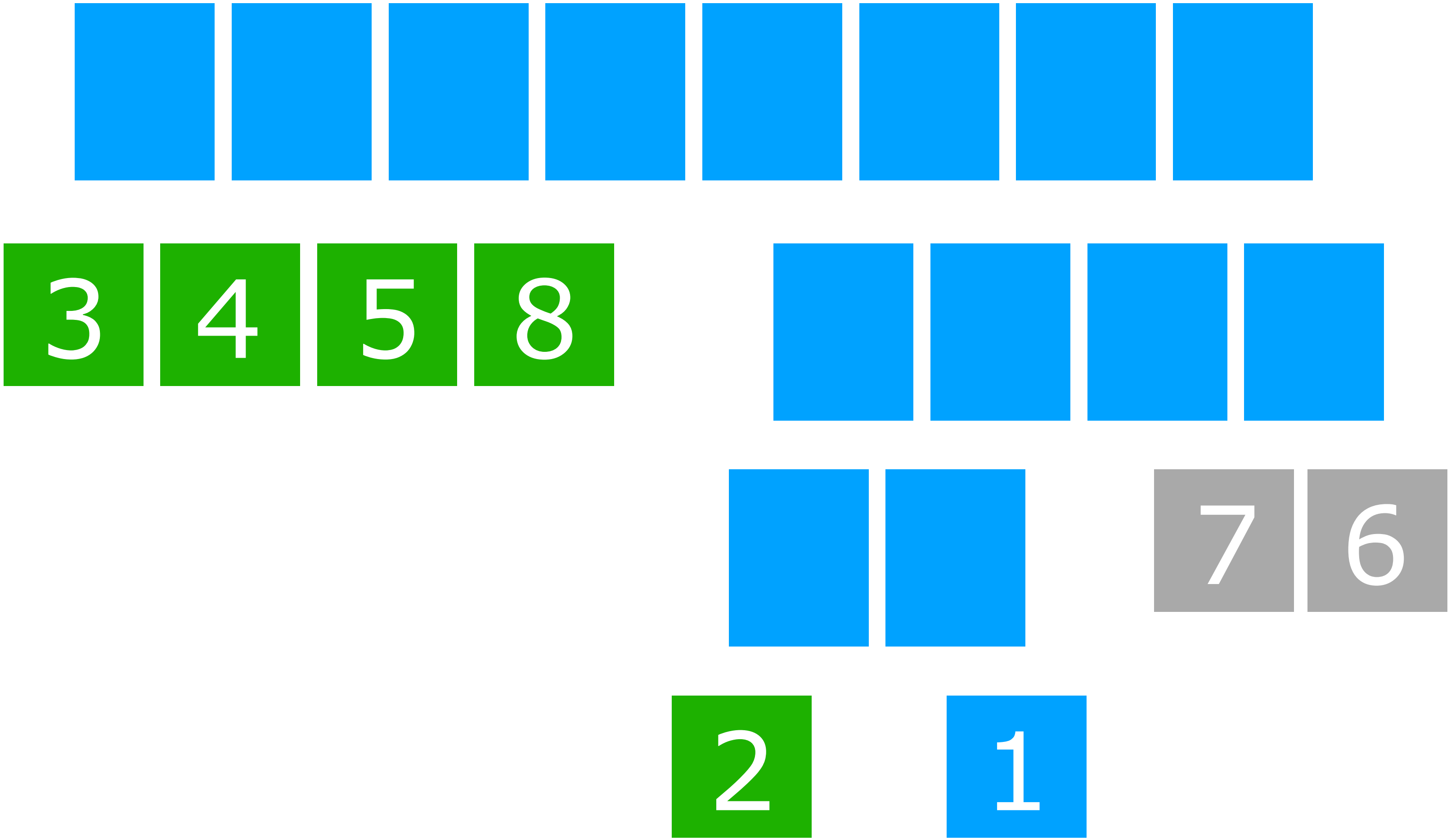

3
4
5
8
2
1
7
6

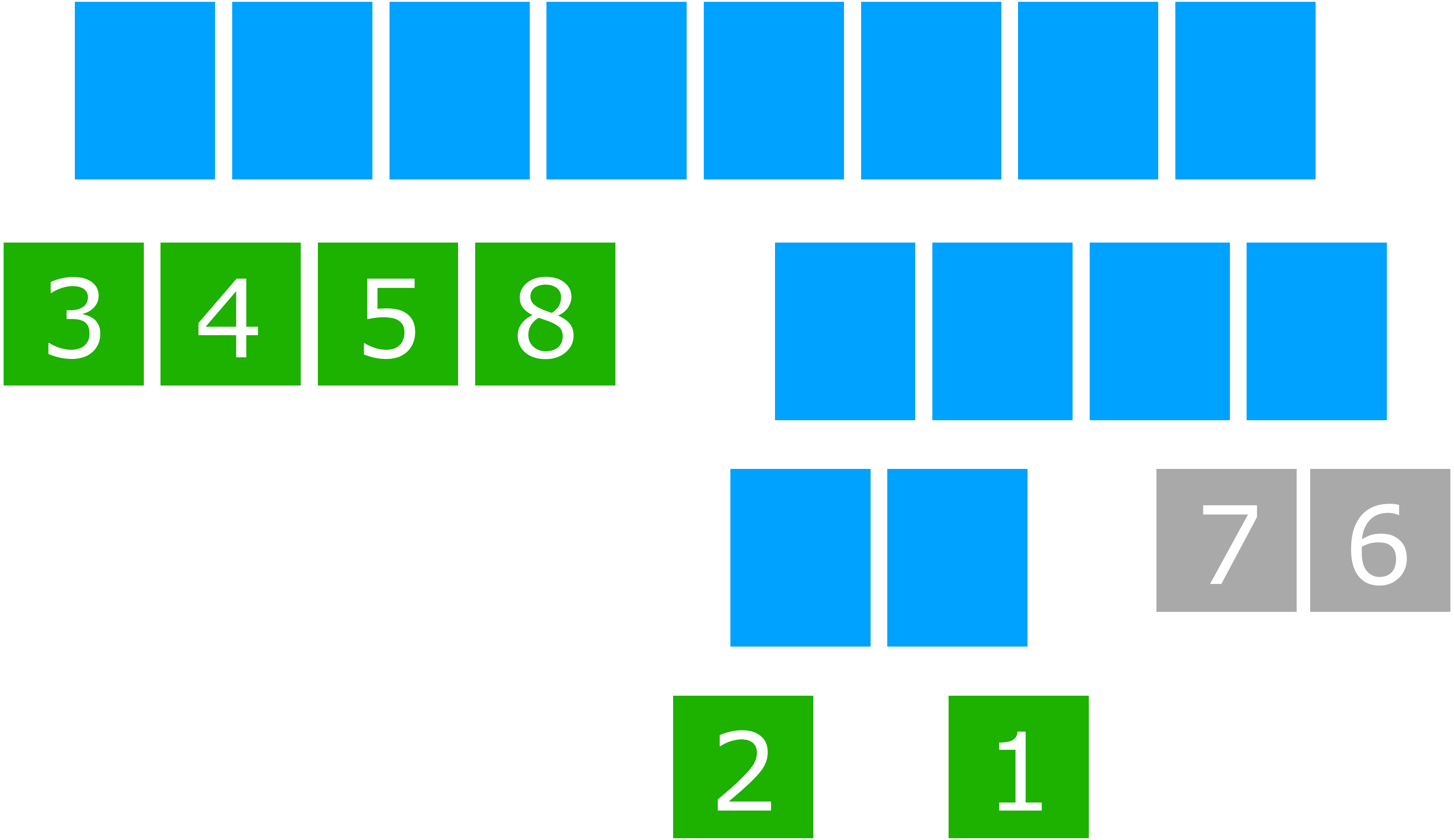

3
4
5
8
2
1
7
6

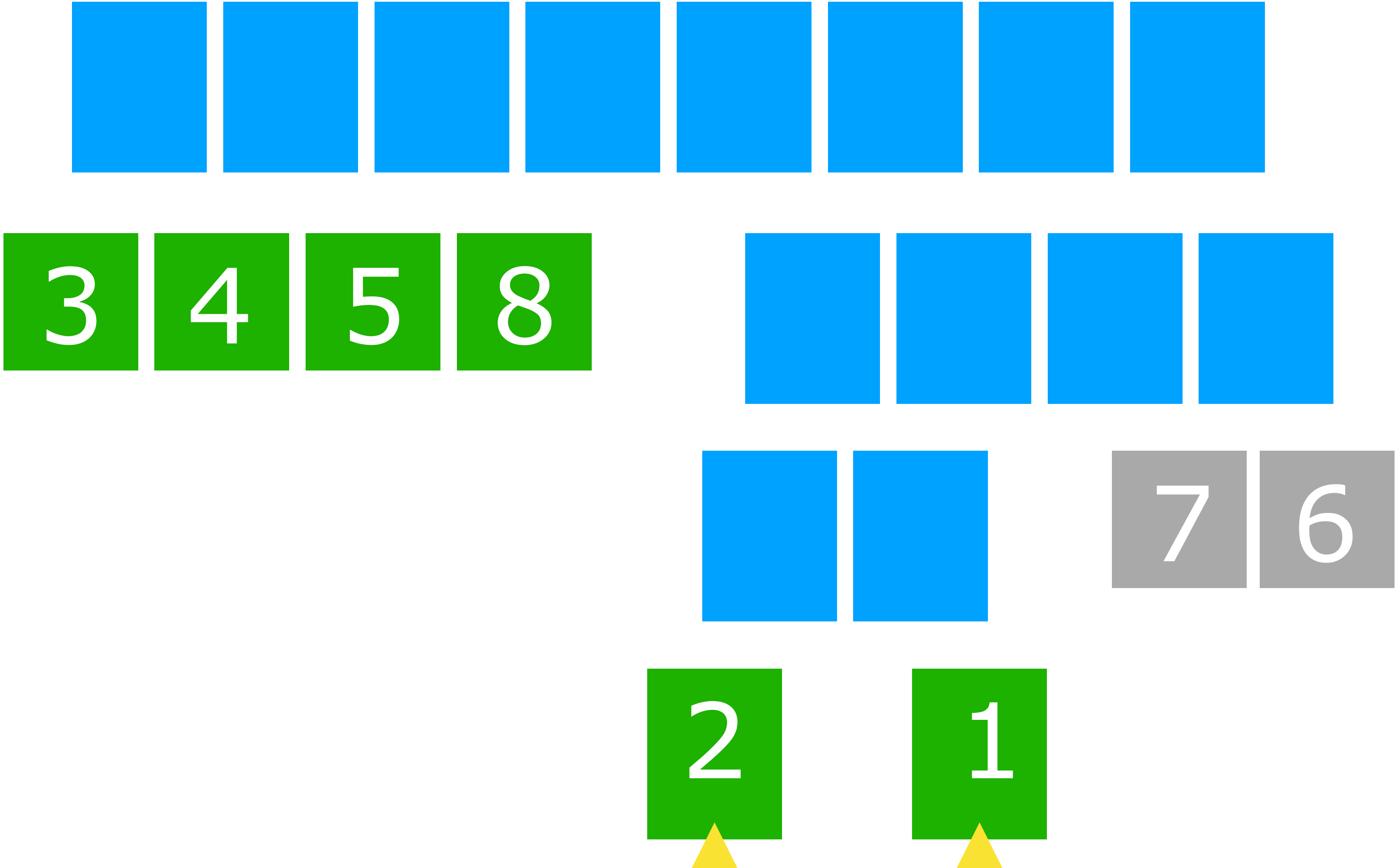

3
4
5
8
7
6
2
1

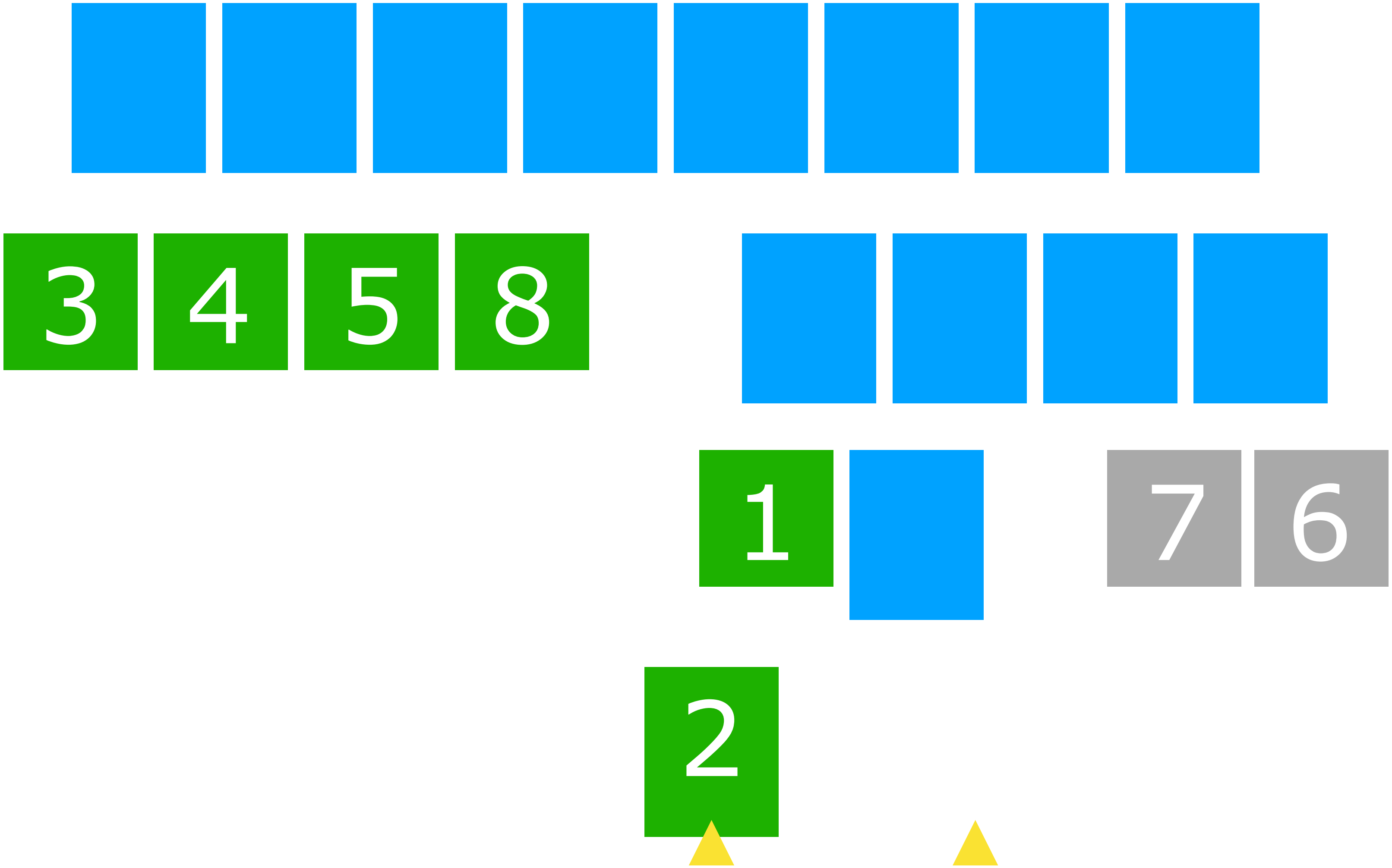

3
4
5
8
7
6
2
1

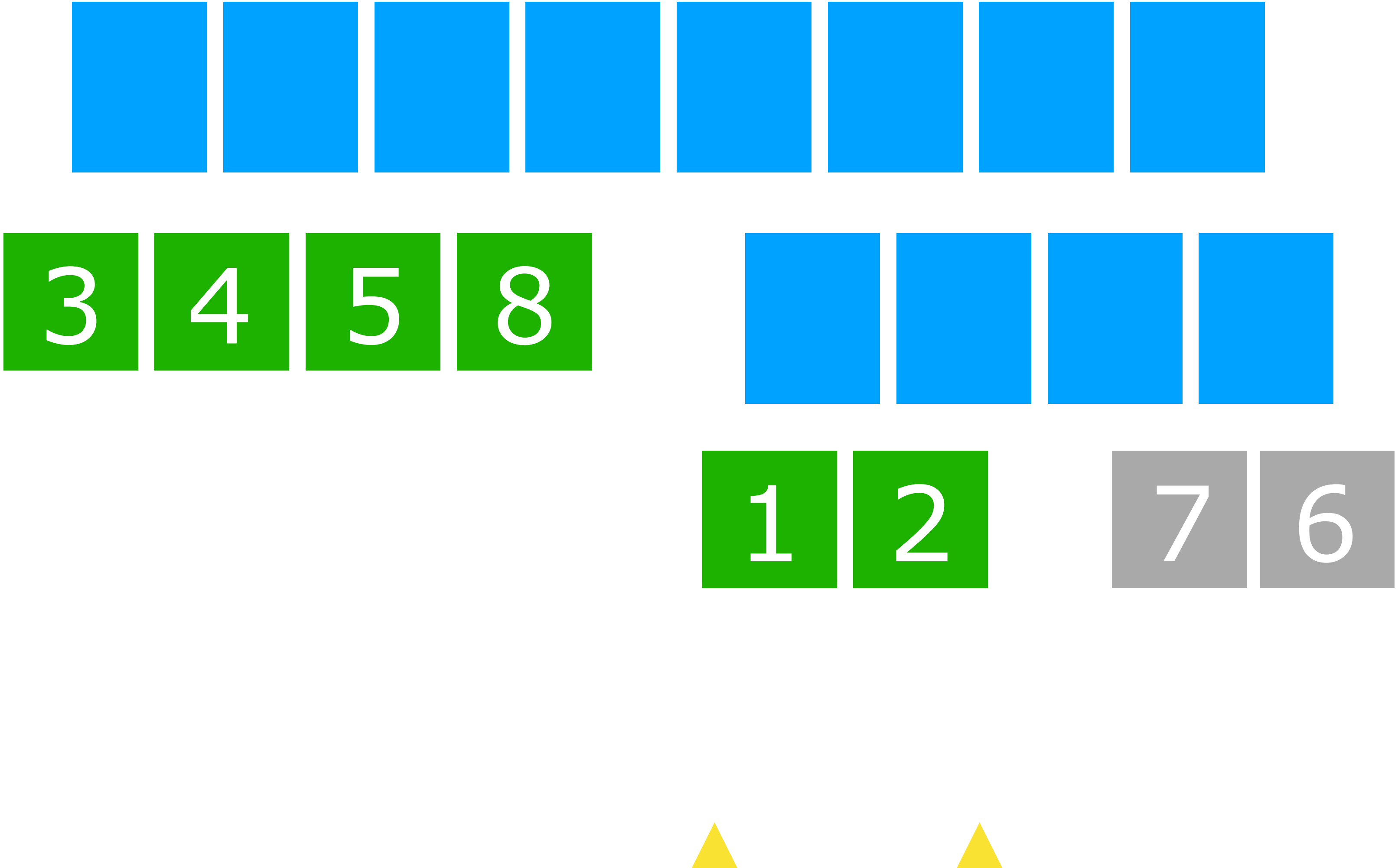

3
4
5
8
7
6
2
1

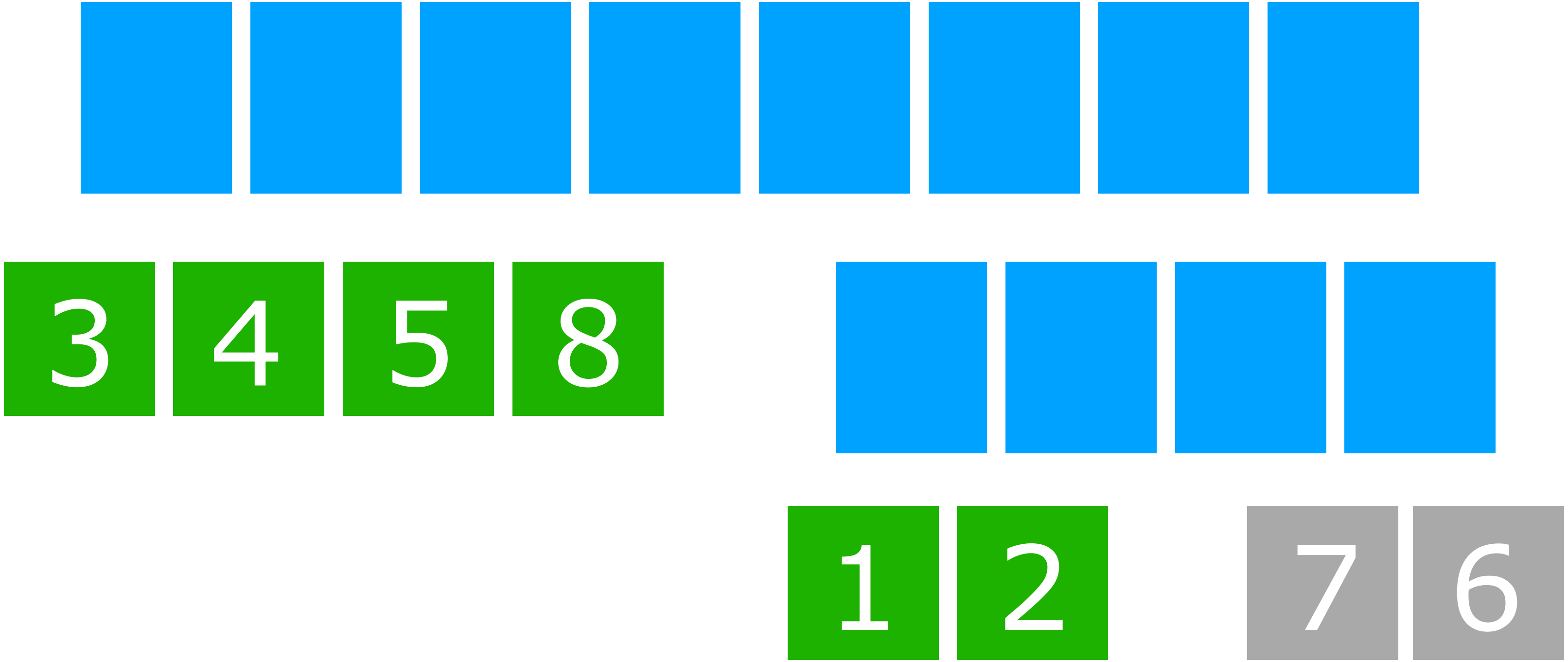

3
4
5
8
7
6
2
1


3
4
5
8
7
6
2
1


3
4
5
8
7
6
2
1


3
4
5
8
1
7
6
2

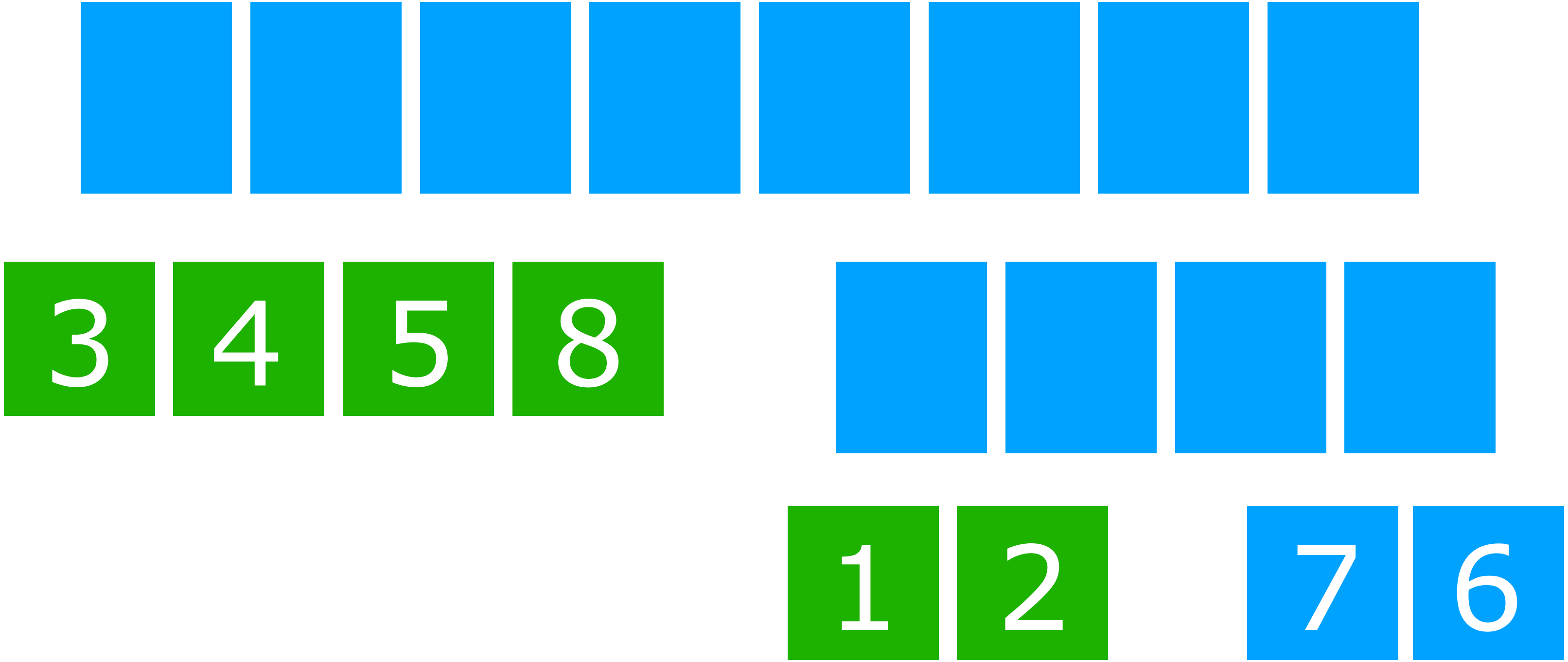

3
4
5
8
1
2
7
6

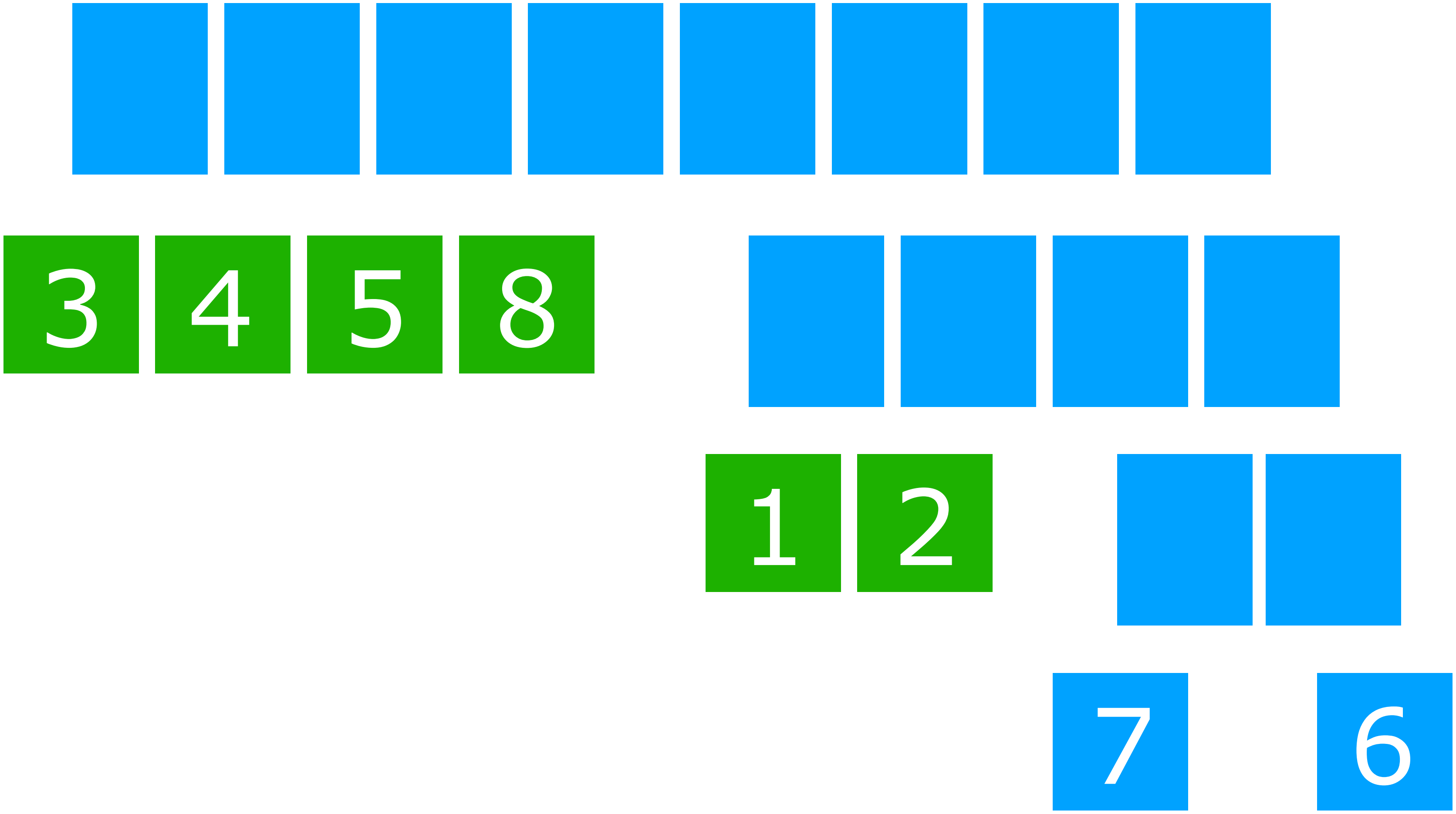


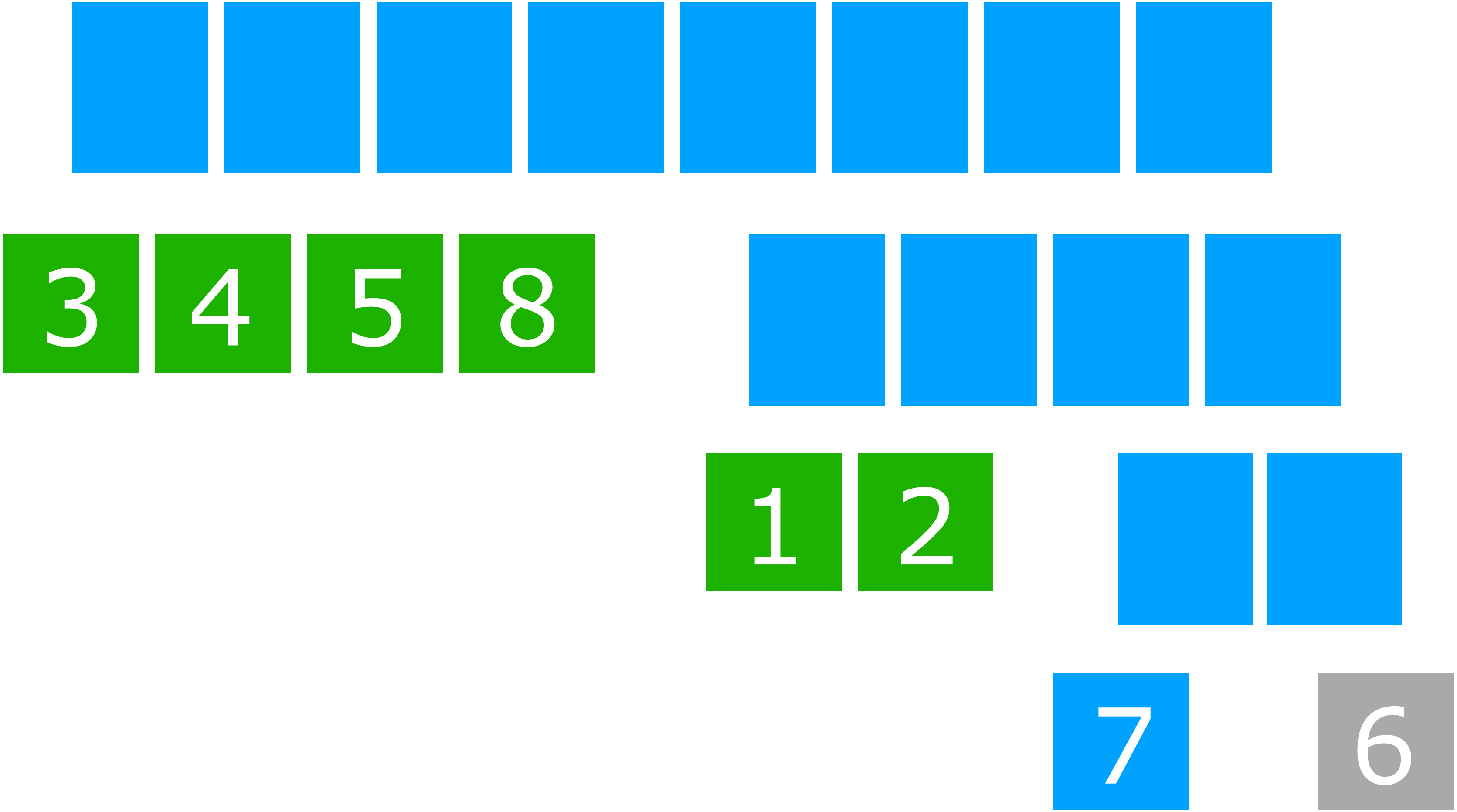
3 4 5 8

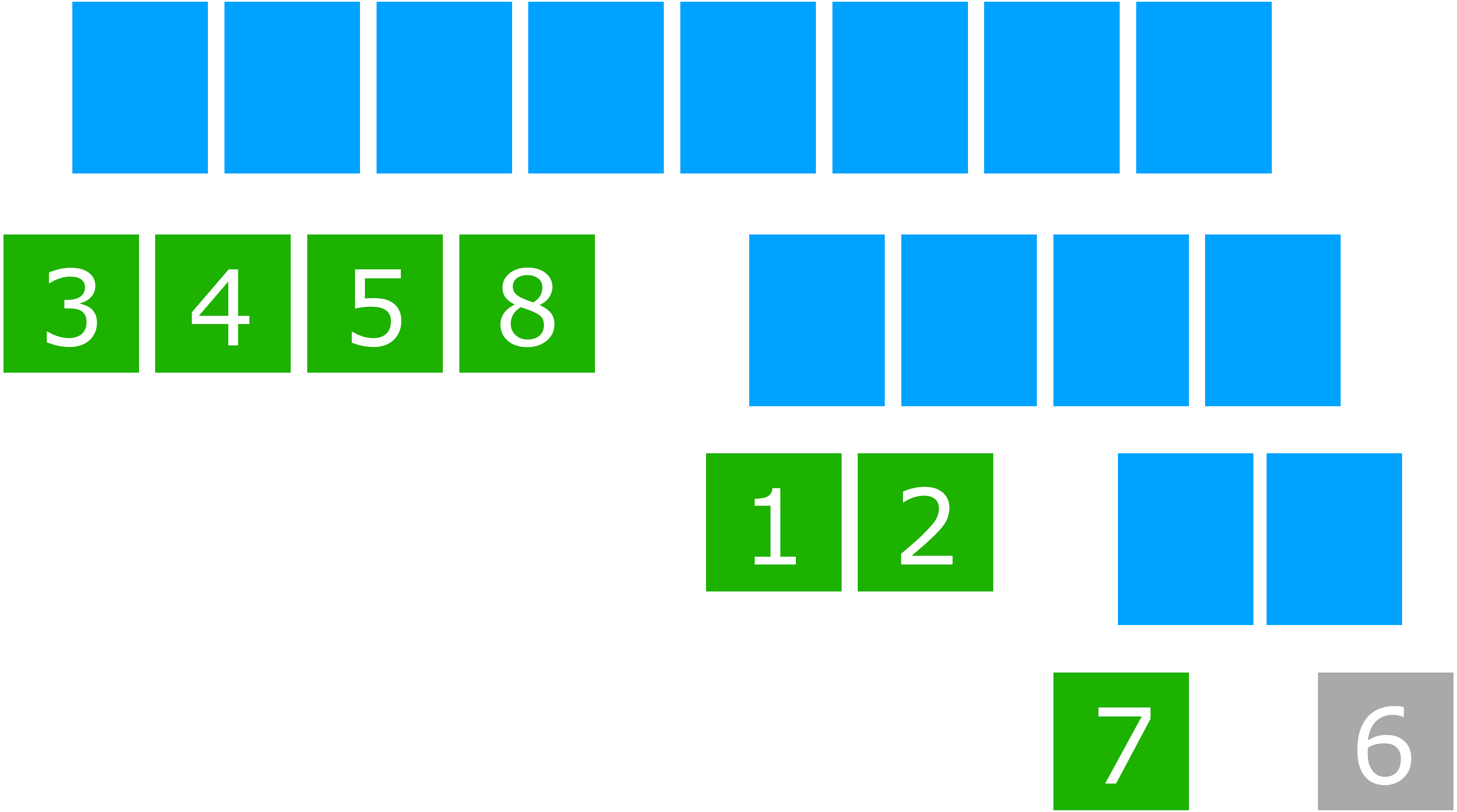
1 2

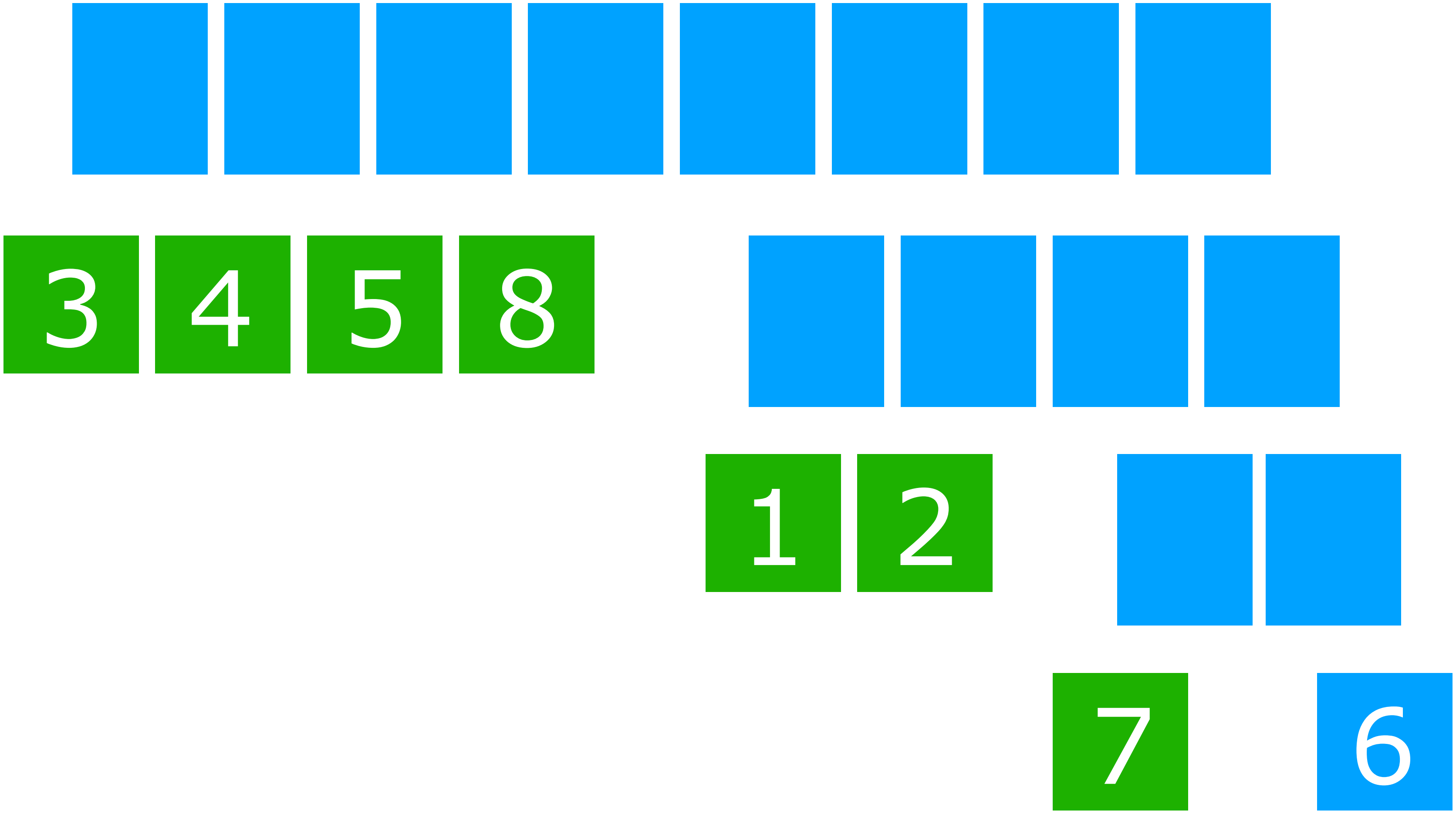
7 6

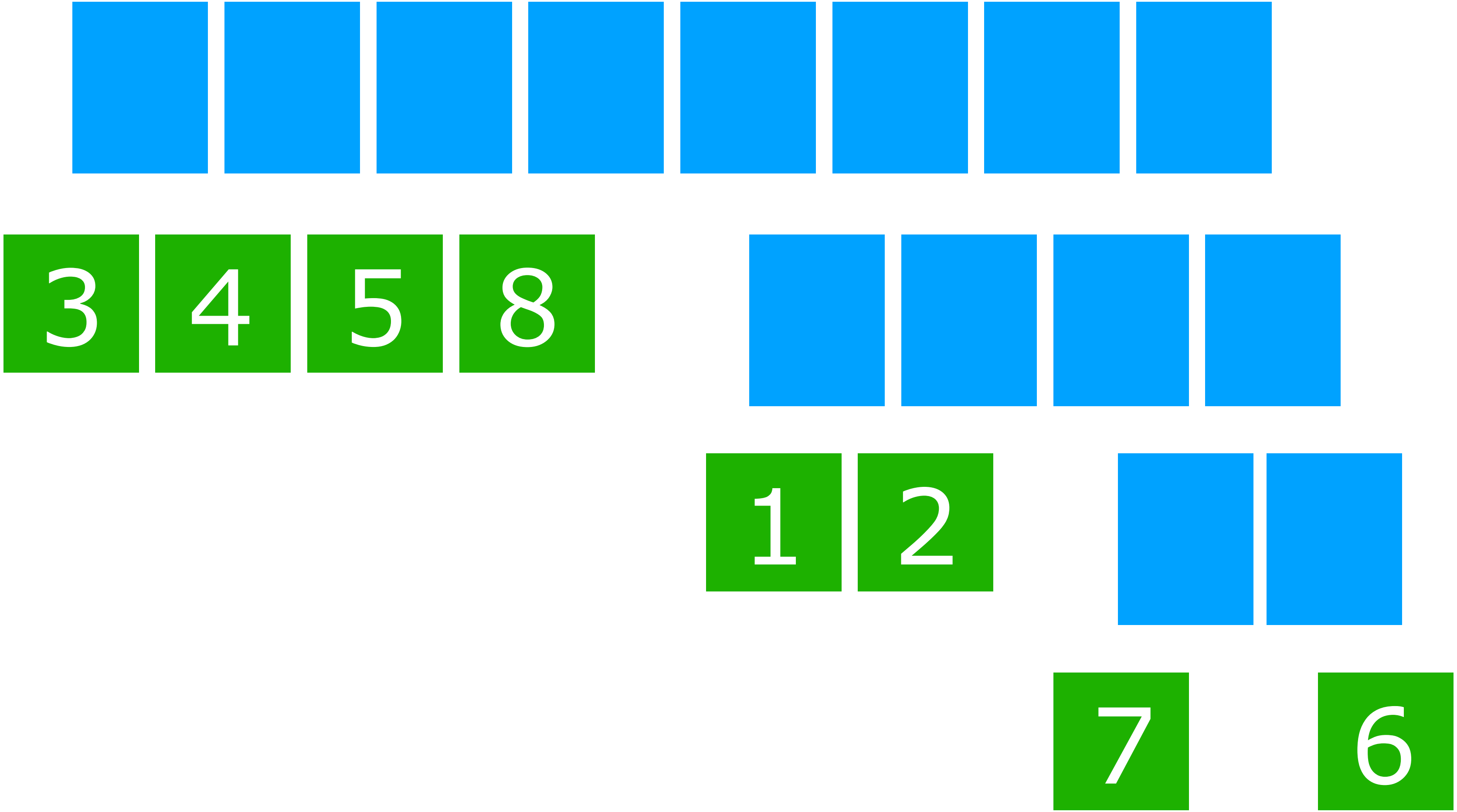

3
4
5
8
1
2
7
6

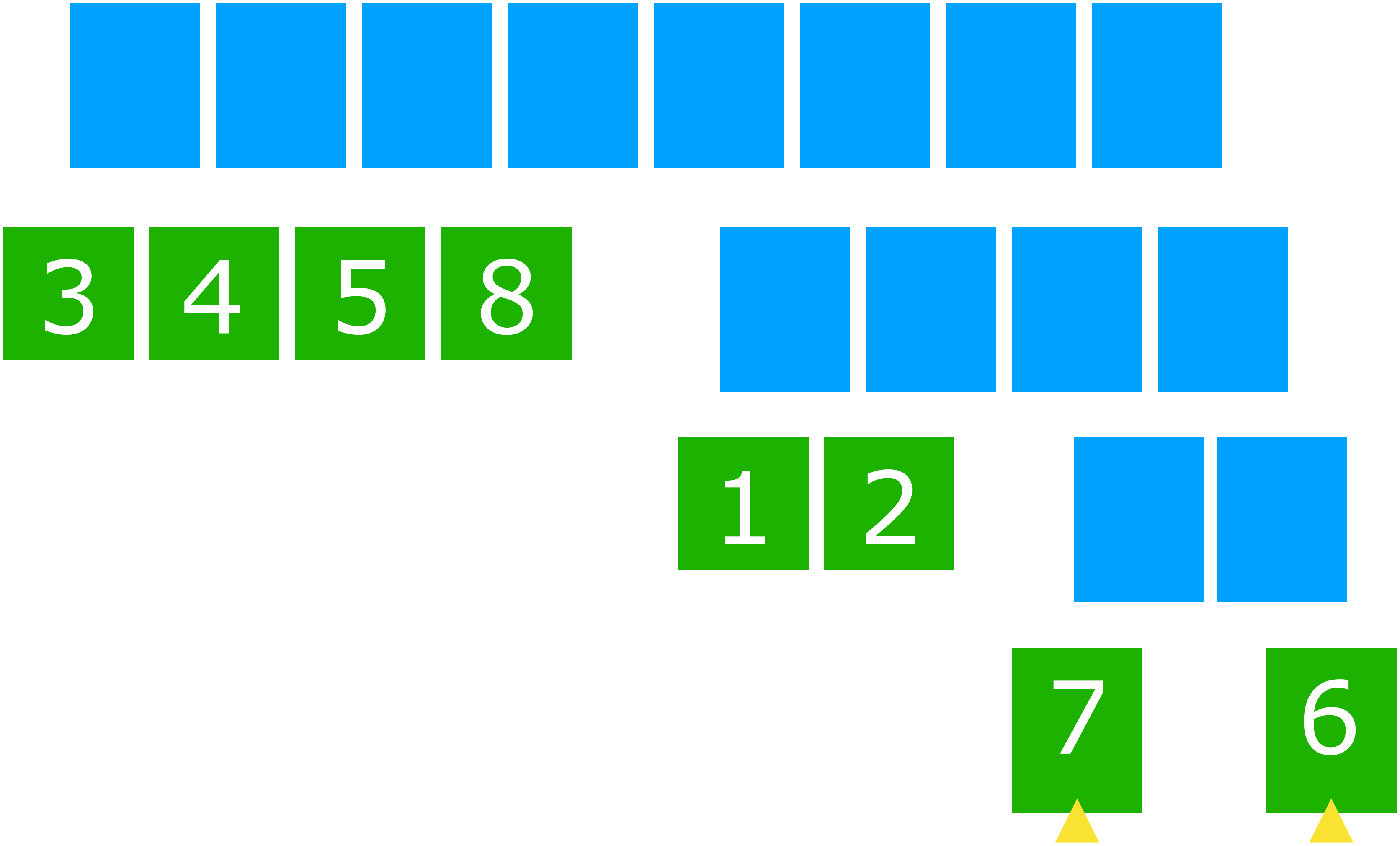

3
4
5
8
1
2
7
6

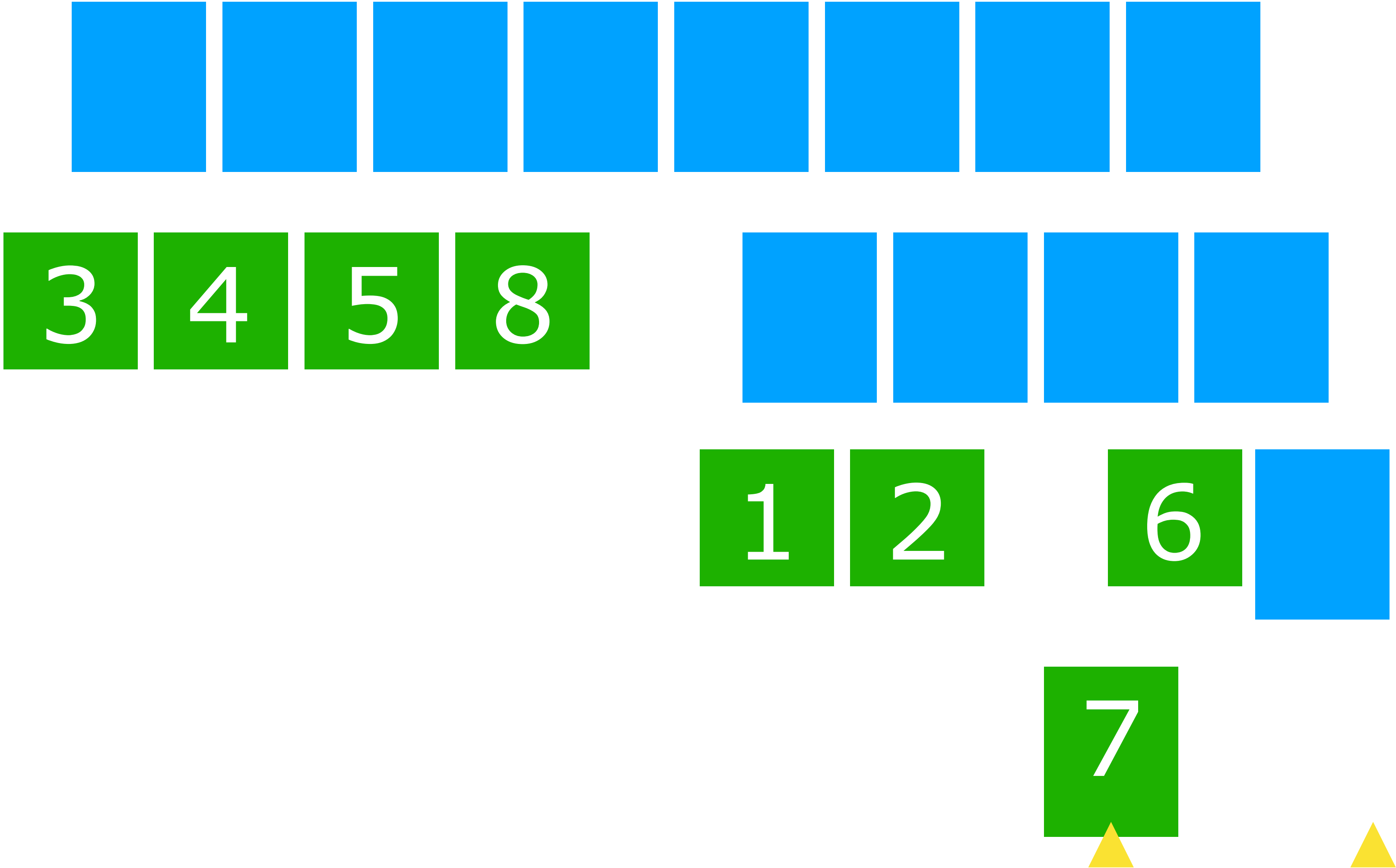

3
4
5
8
1
2
7
6

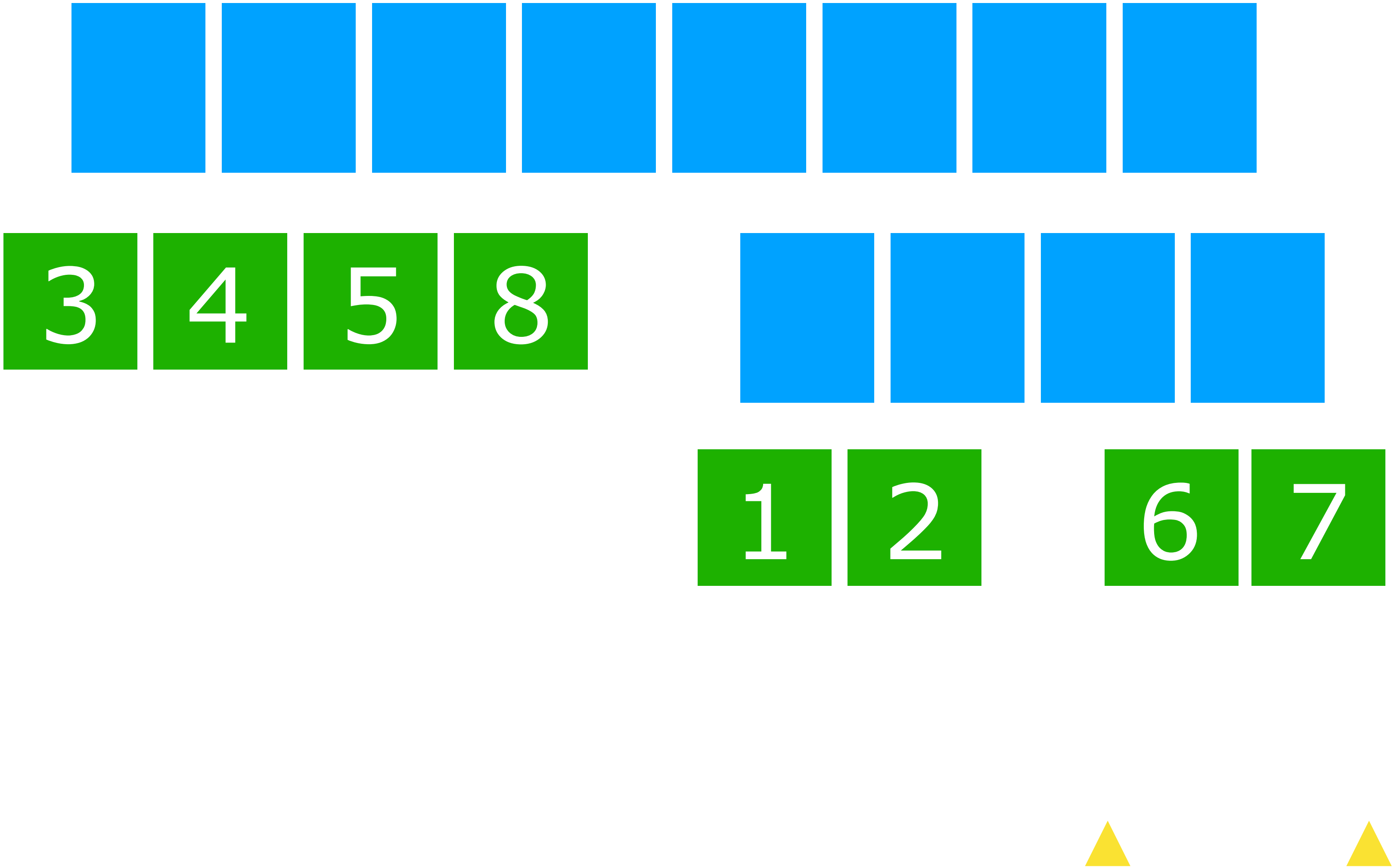

3
4
5
8
1
2
7
6

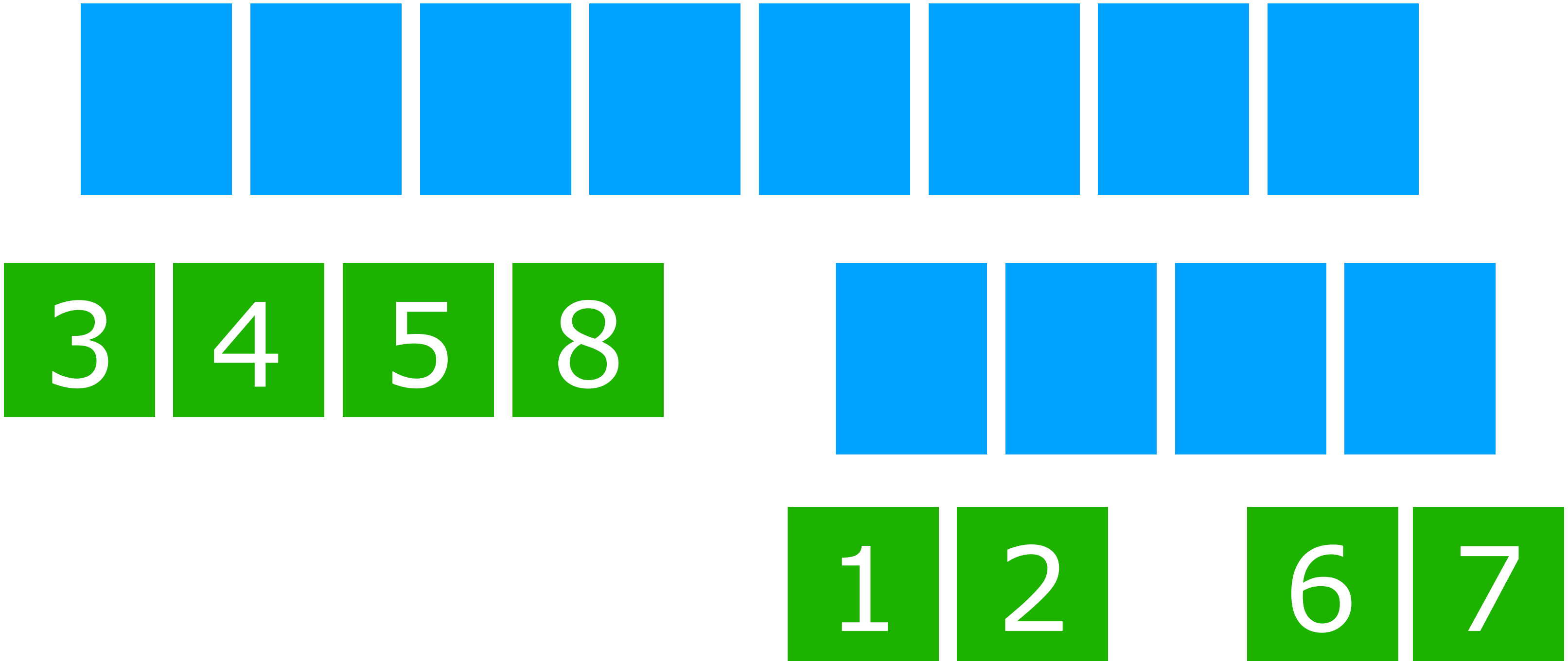

3
4
5
8
1
2
7
6


3
4
5
8
1
2
7
6


3
4
5
8
1
2
7
6


3
4
5
8
1
2
6
7

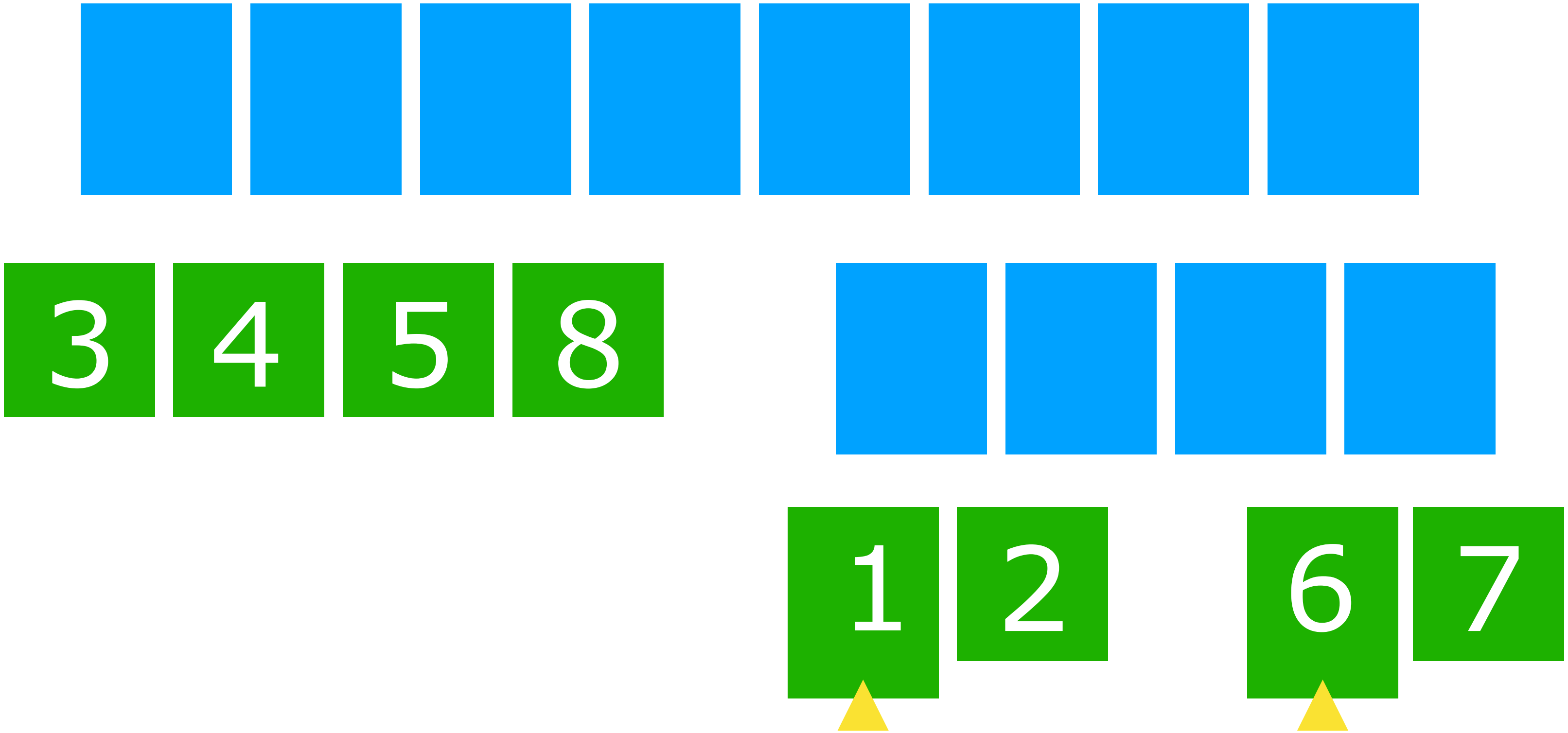

3
4
5
8
1
2
6
7

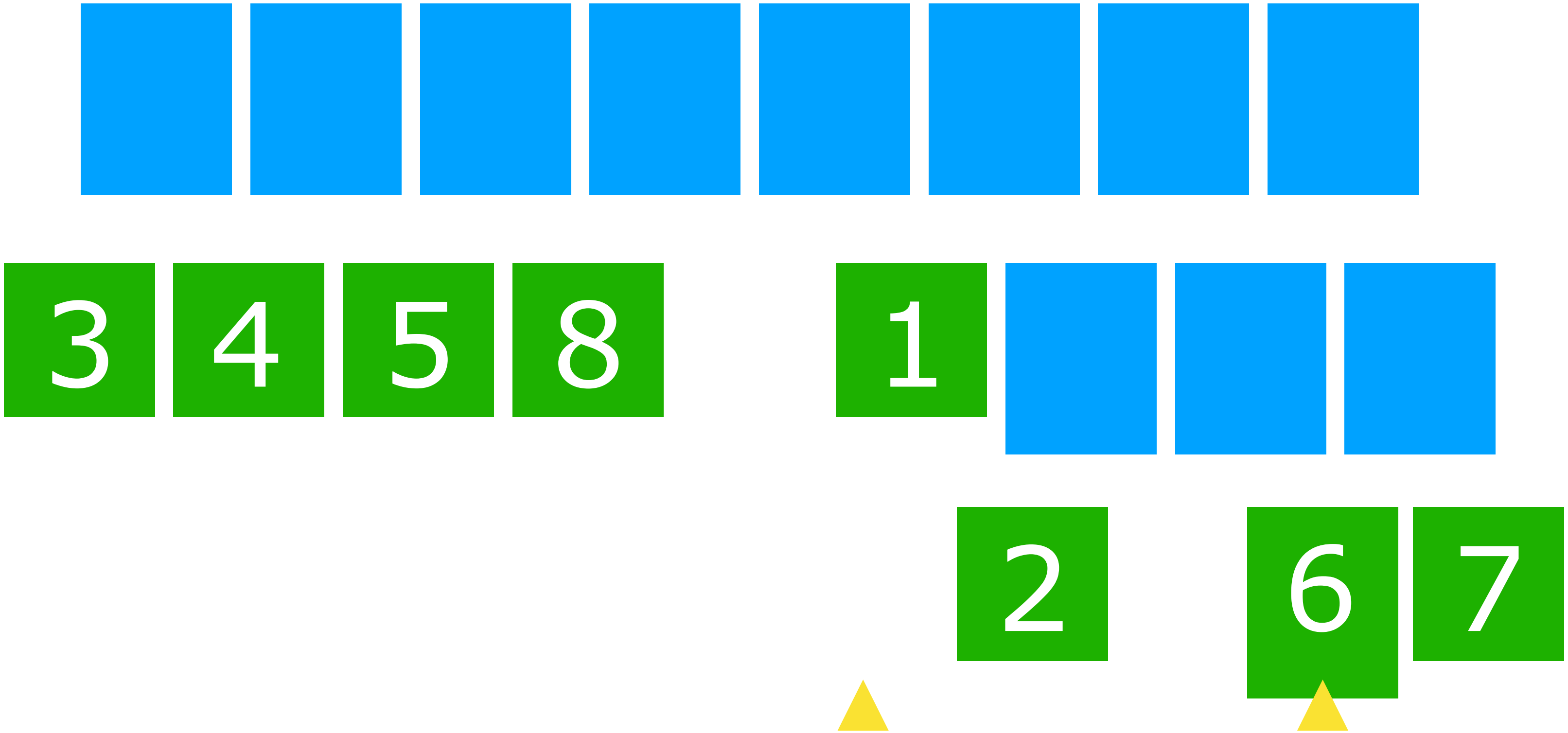


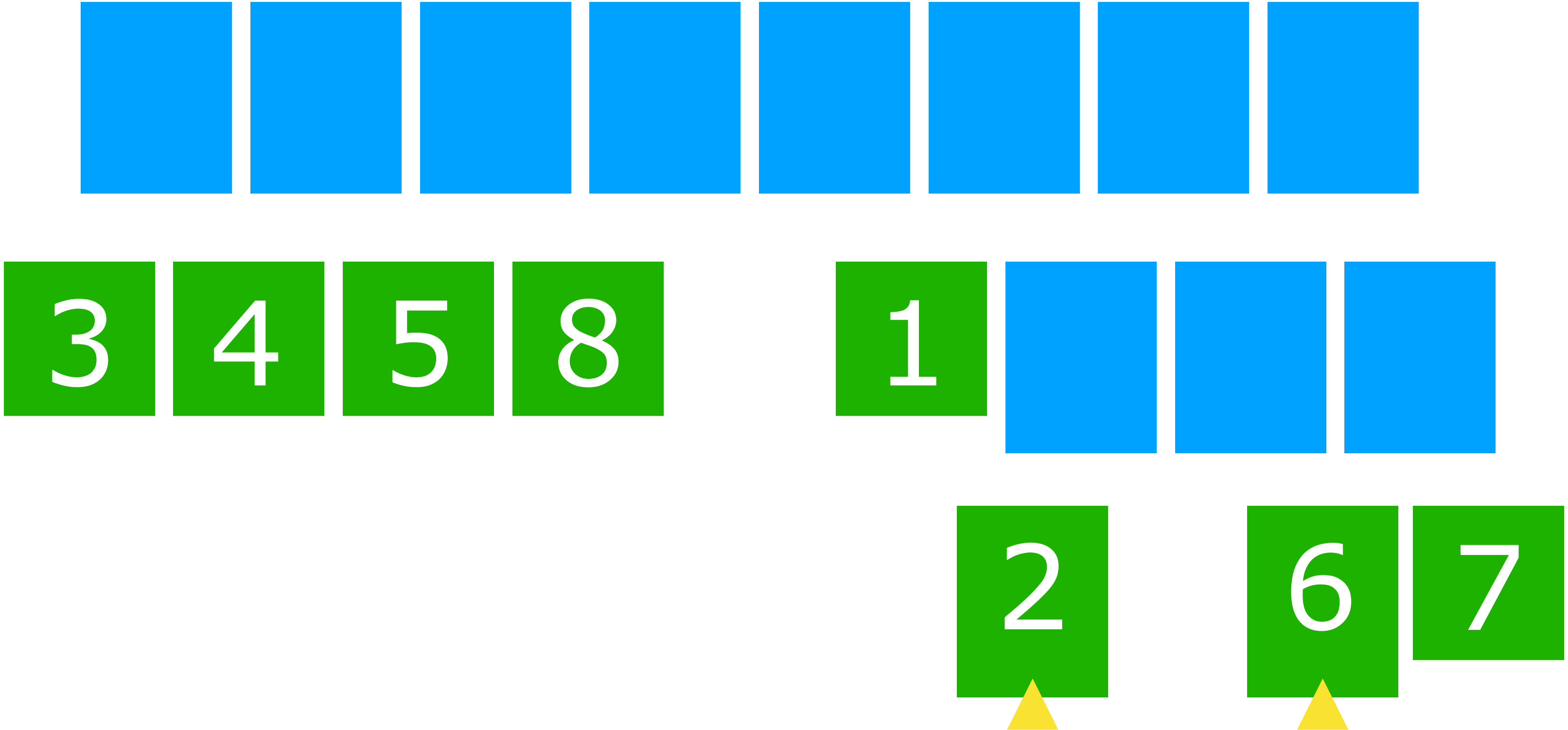
3 4 5 8

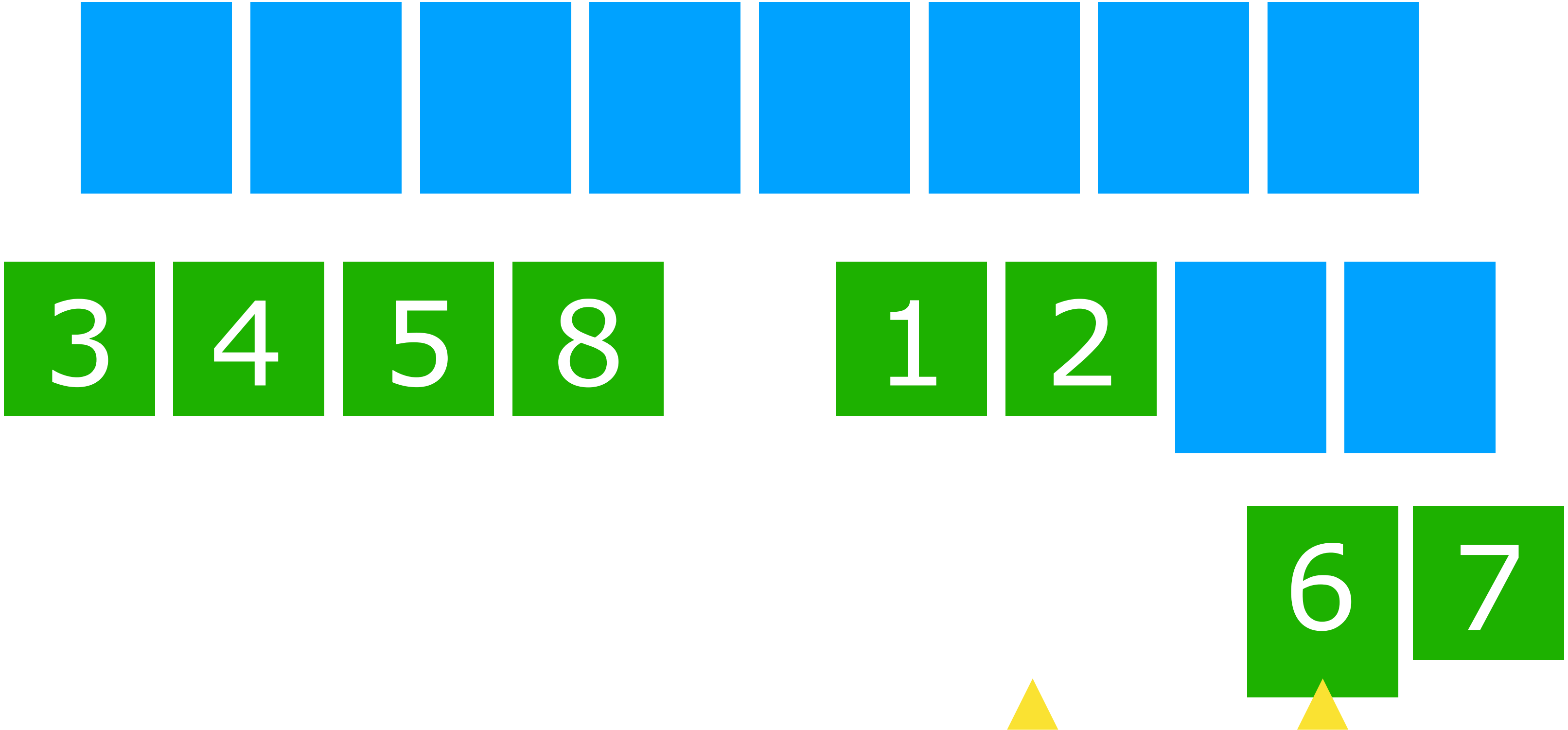
1 2

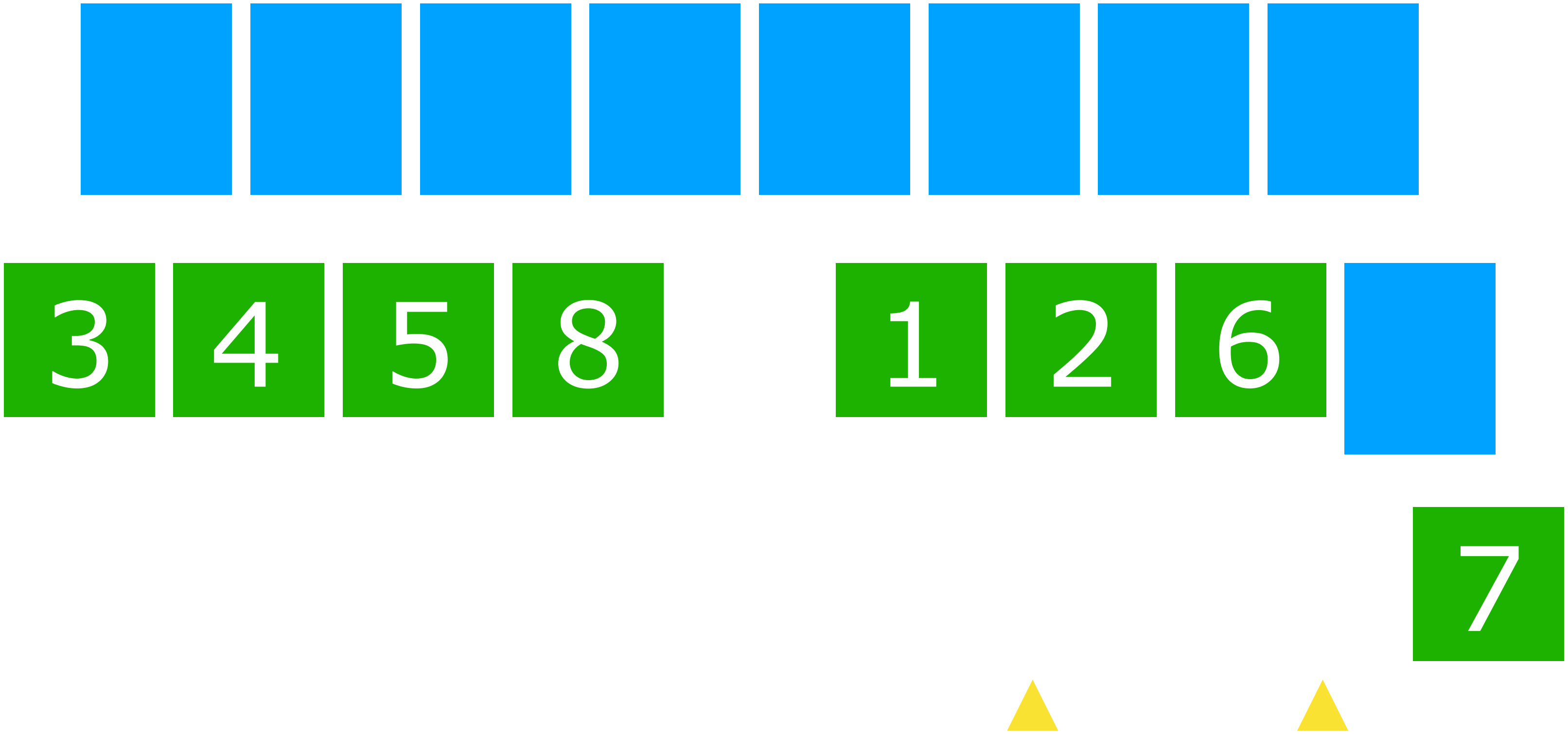
6 7

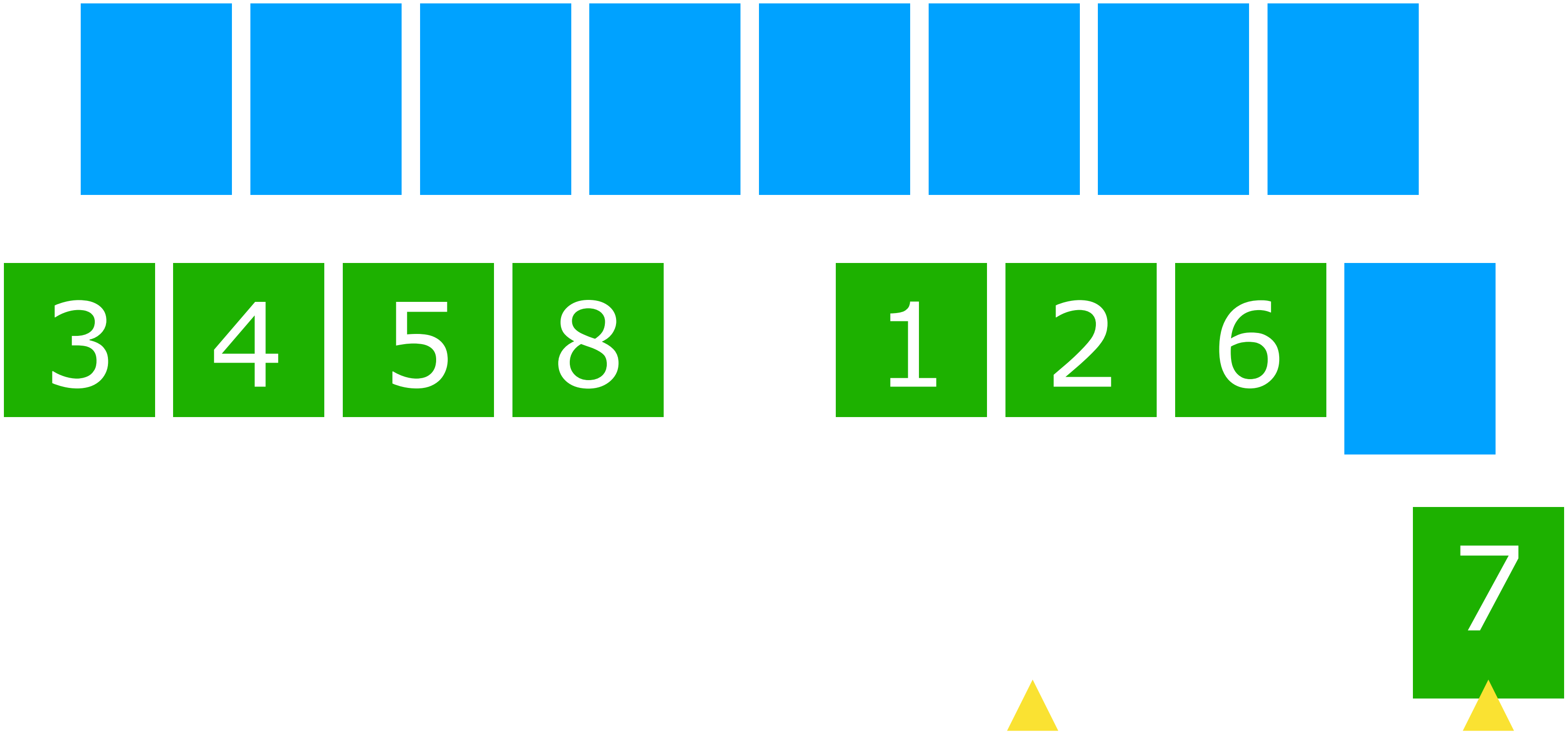

3
4
5
8
1
2
6
7

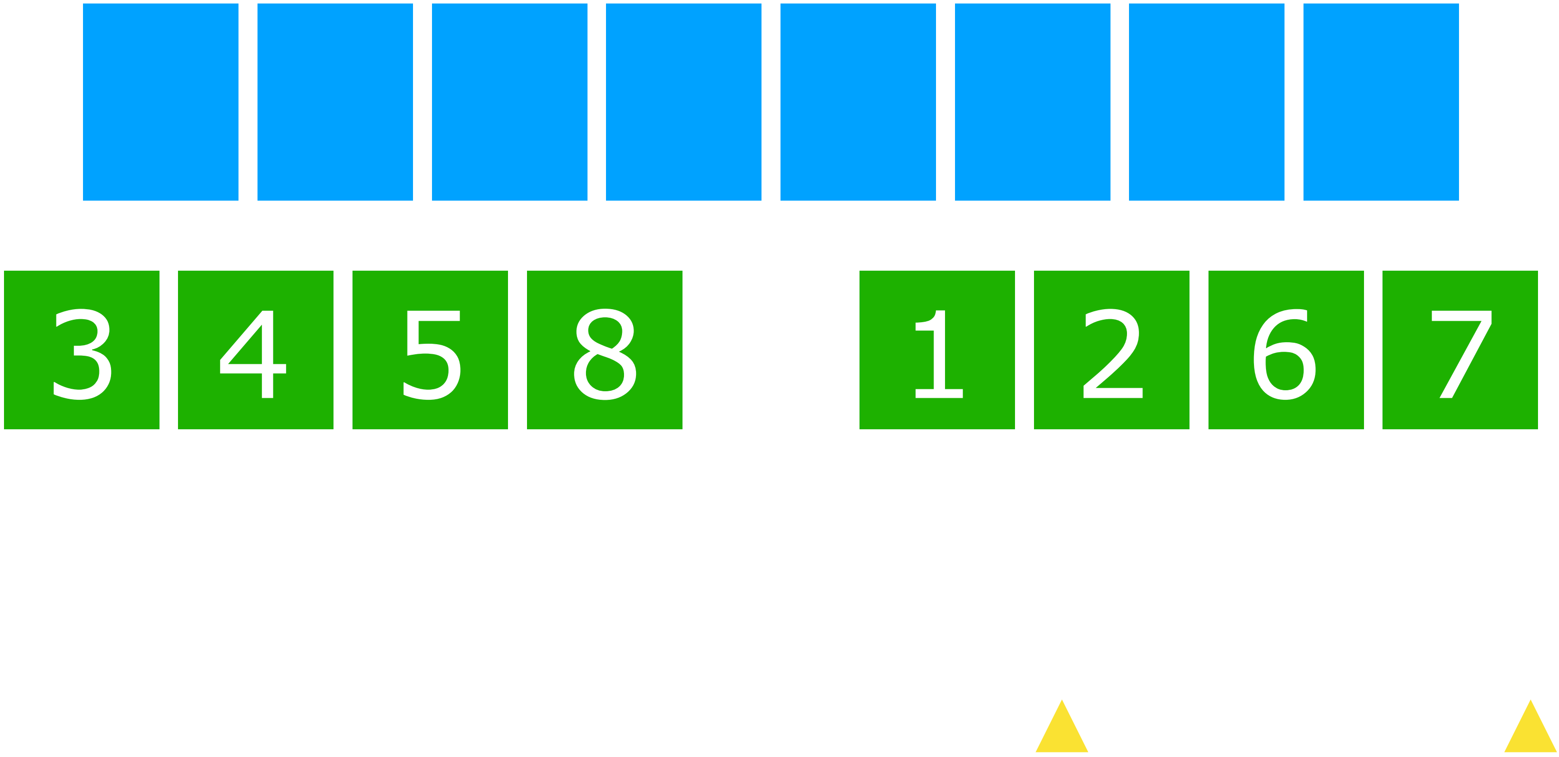

3
4
5
8
1
2
6
7


3
4
5
8
1
2
6
7


3
4
5
8
1
2
6
7

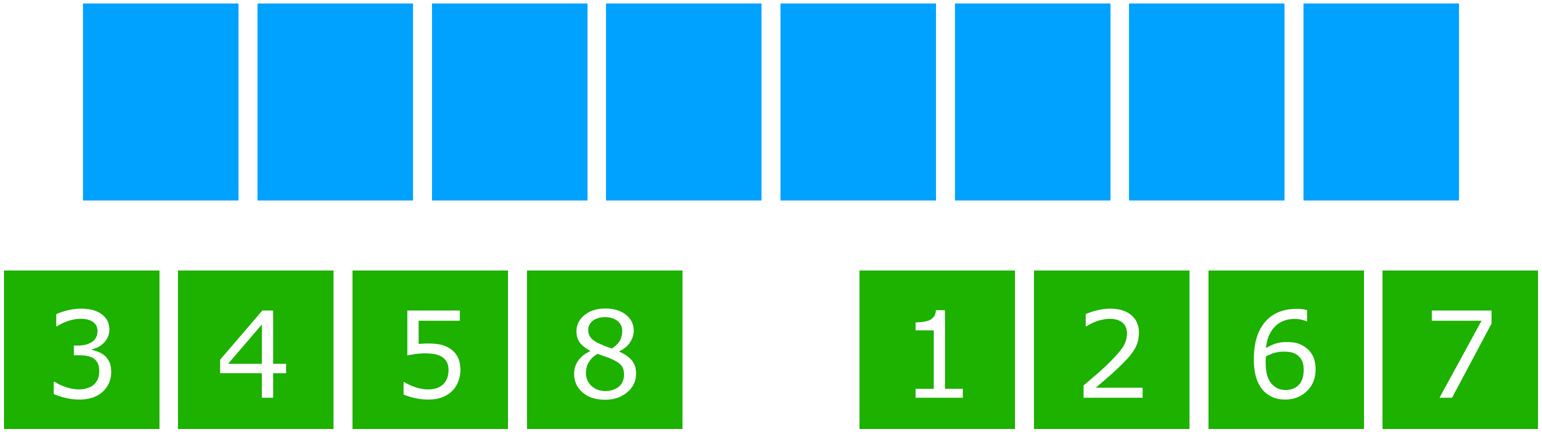

3
4
5
8
1
2
6
7


3
4
5
8
1
2
6
7



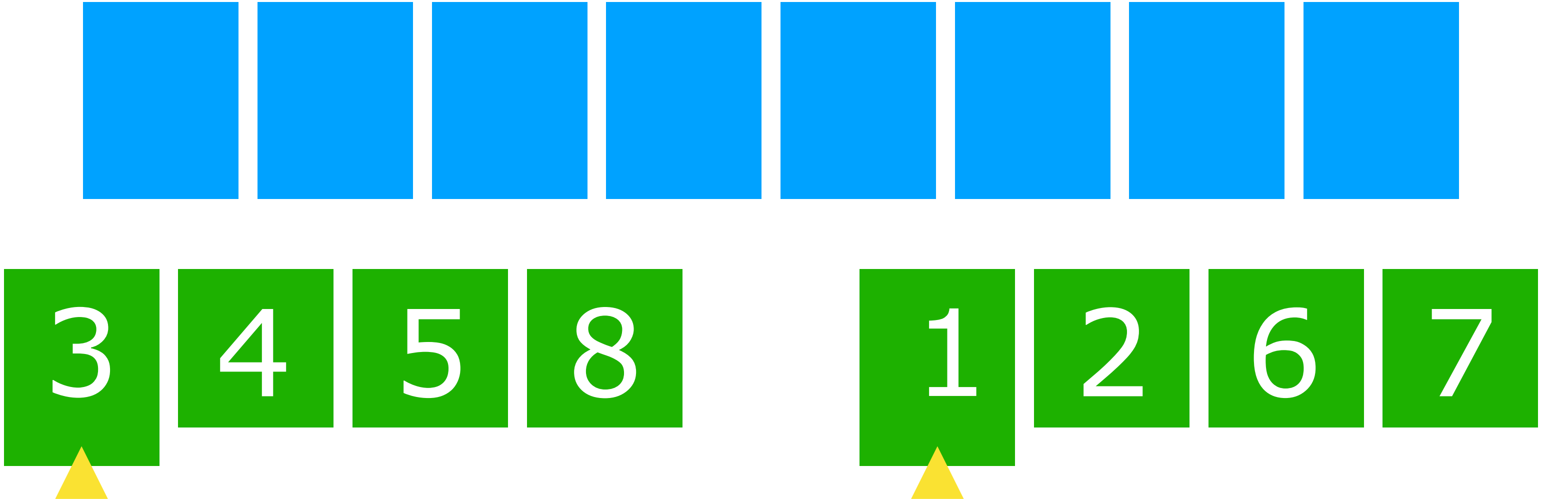
3 4 5 8

1 2 6 7



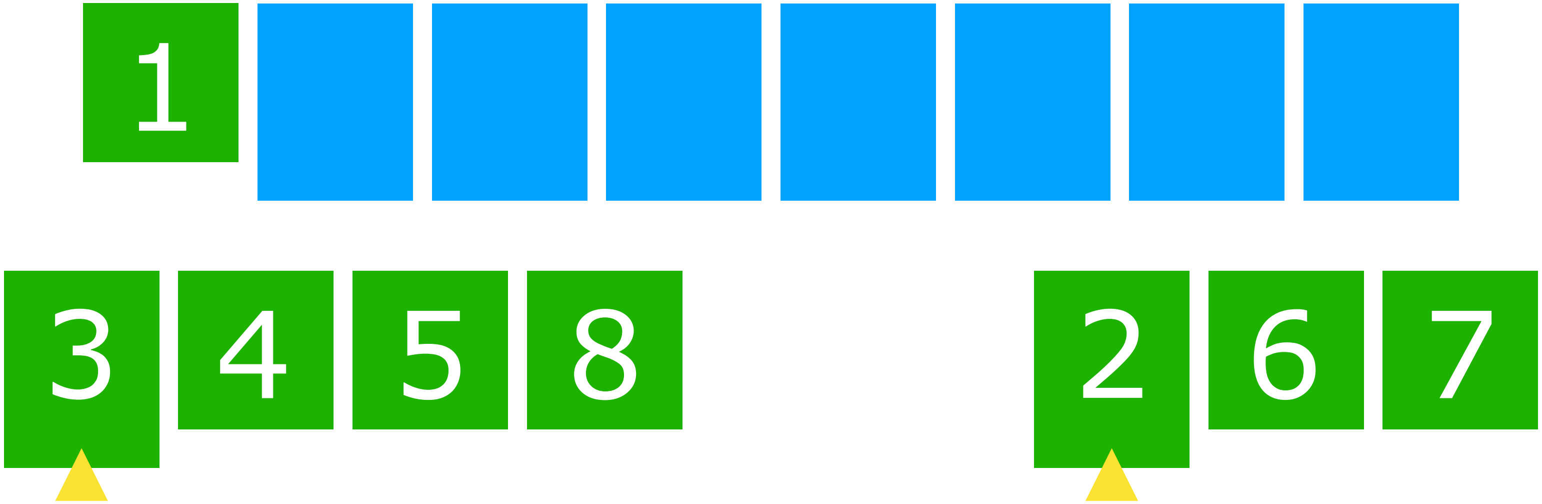
3 4 5 8

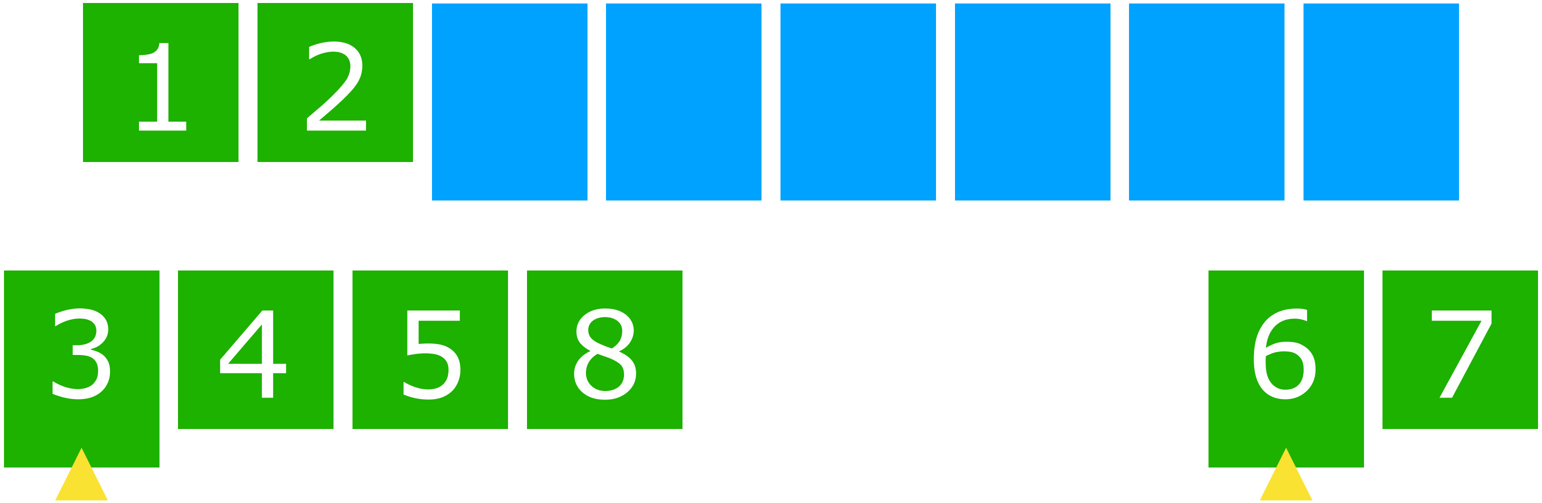
1 2 6 7

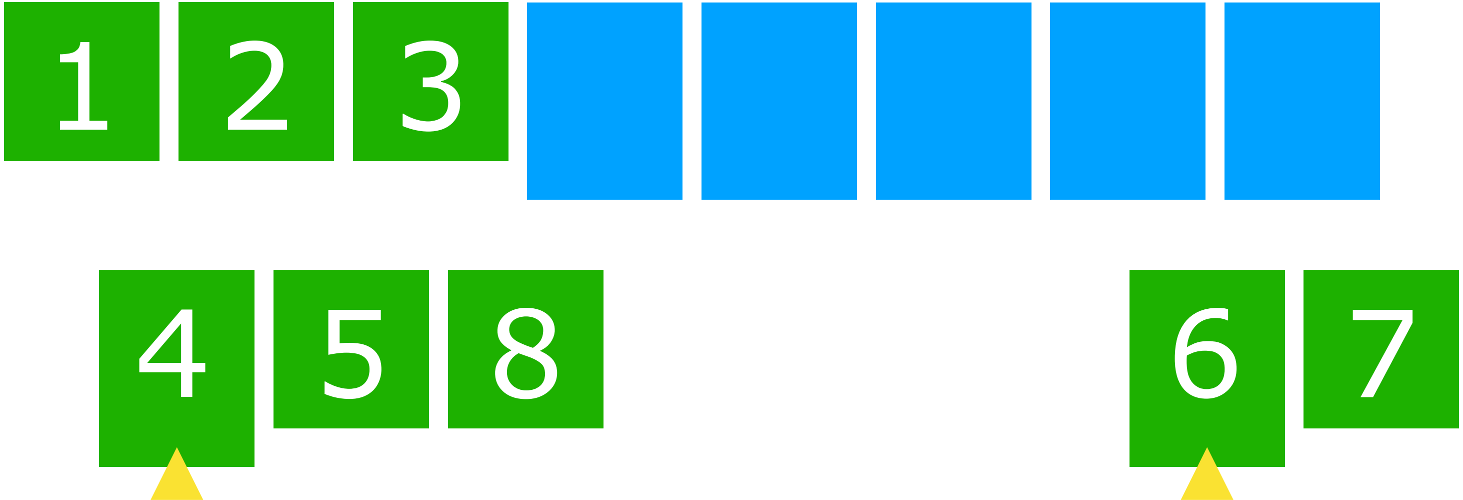


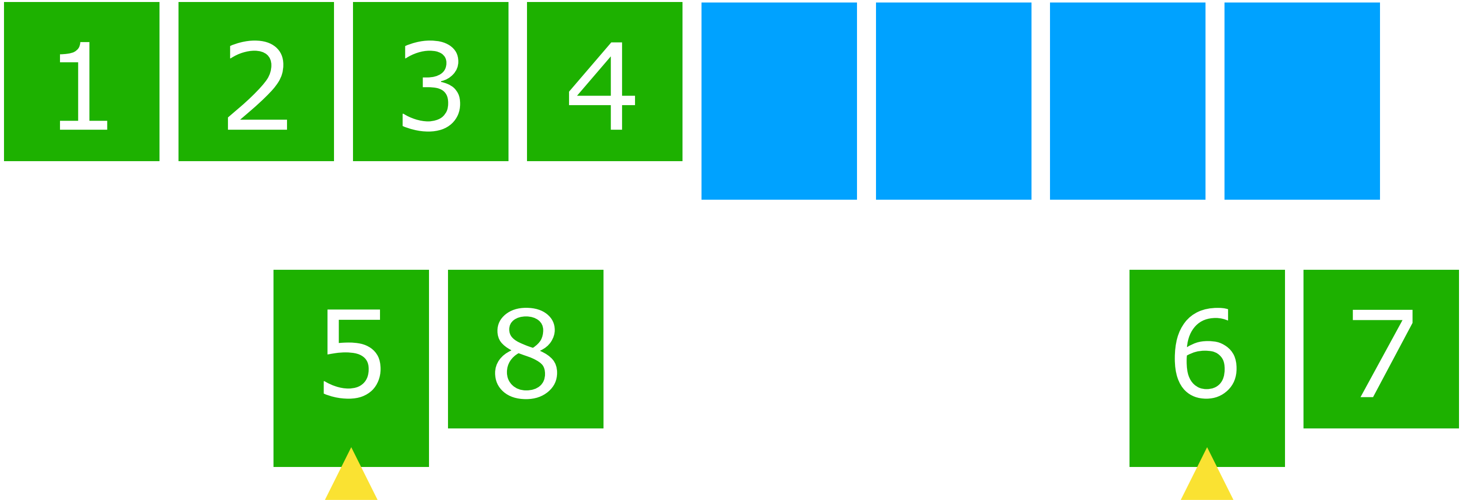
3 4 5 8

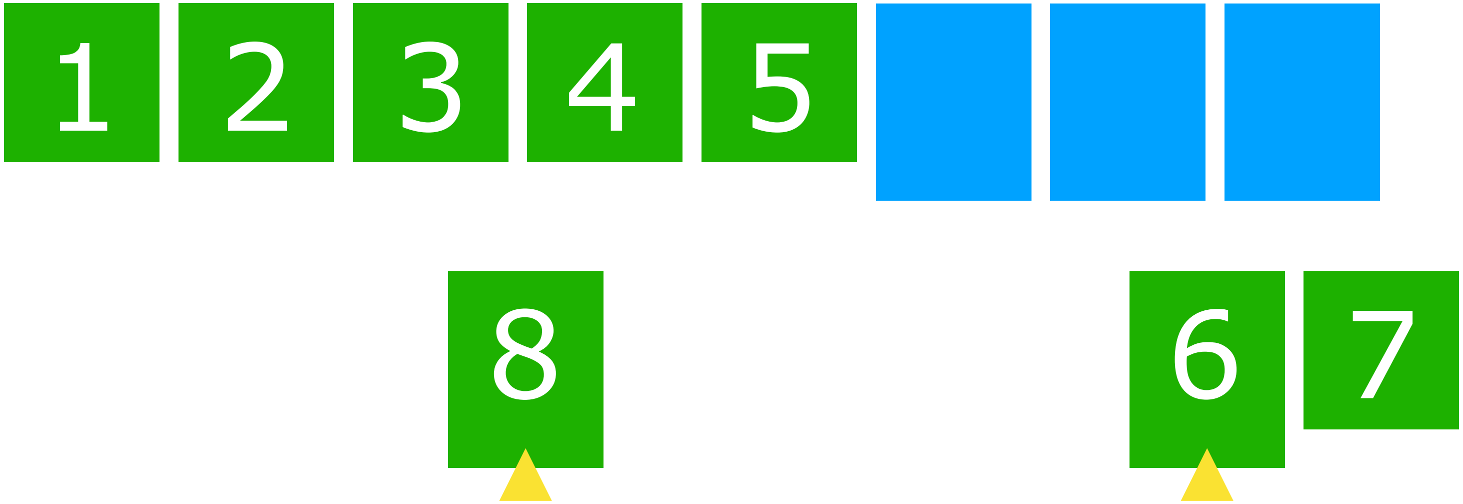
1 2 6 7

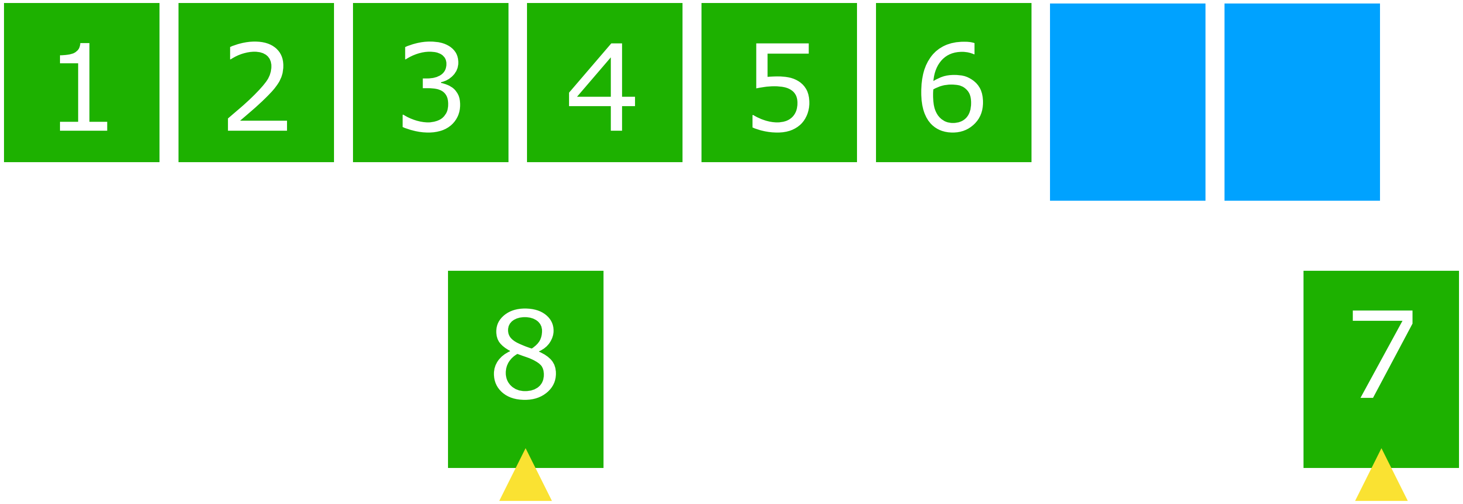

1
3
4
5
8
2
6
7

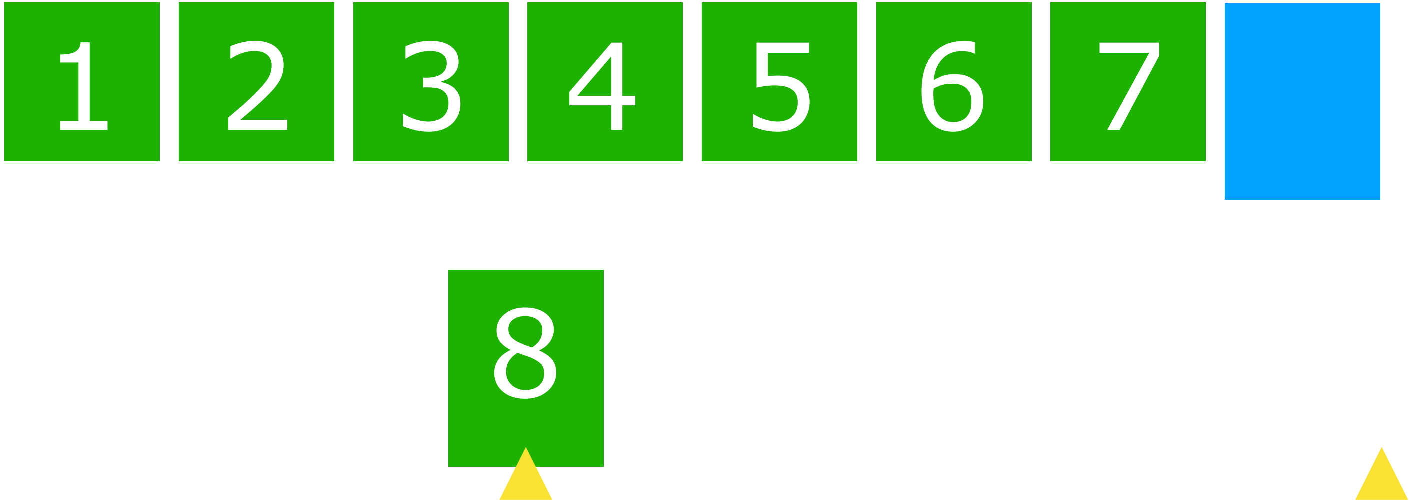

1
2
3
4
5
8
6
7

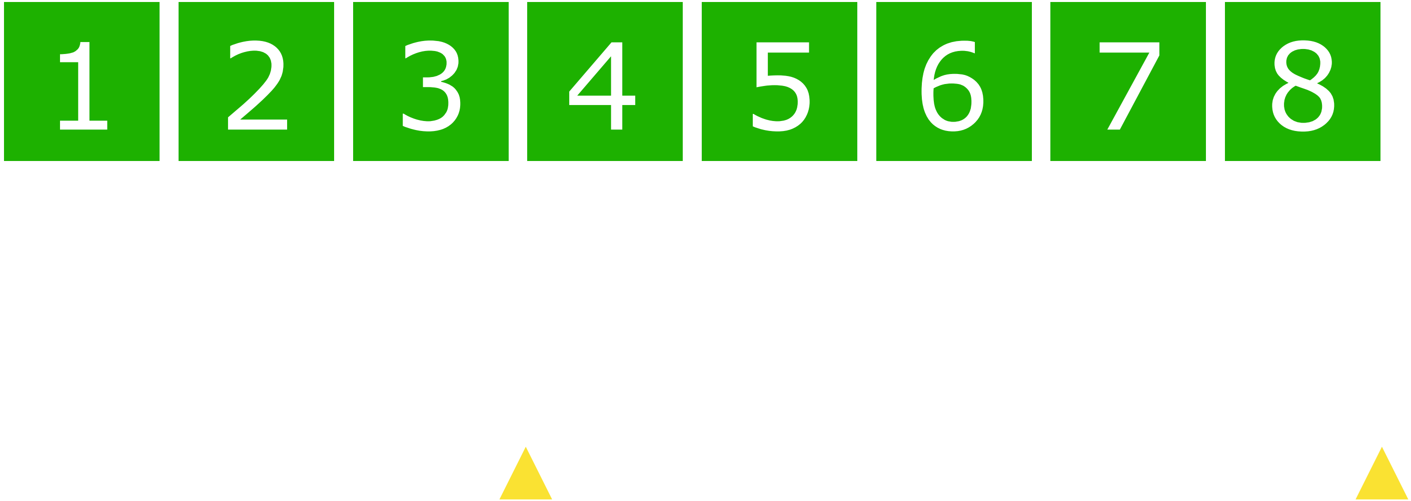

1
2
3
4
5
8
6
7


1
2
3
4
5
8
6
7


1
2
3
4
5
8
6
7


1
2
3
4
5
6
8
7


1
2
3
4
5
6
7
8



1	2	3	4	5	6	7	8

1 2 3 4 5 6 7 8

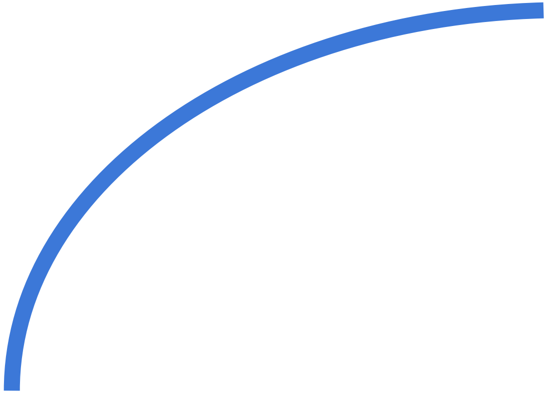
1	2	3	4	5	6	7	8

O(nlog$_2$(n)) — "worst case" definition

In the worst case, I need to do about log$_2$(n) steps to find my solution.

$O(n\log_2(n))$ — "scaling" definition

I don't need to take another step in my algorithm until I double my input.



$\Omega(n\log_2(n))$ — "best case" definition

In the best case, I need to do about $\log_2(n)$ steps to find my solution.

Sort

Summary

Selection Sort	Bubble Sort	Merge Sort
$O(n^2)$	$O(n^2)$	$O(n\log n)$
$\Omega(n^2)$	$\Omega(n)$	$\Omega(n\log n)$

Recursion Functions

Call yourself… within yourself. (Crazy!)

Two Parts:

1. Base Case
2. Recursive Call

Factorial

Write a recursive function `factorial` that computes the factorial of a number n. Note that 0! = 1.

Factorial

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

Factorial

factorial(1)

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

Factorial

factorial(1)

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

false

Factorial

factorial(1)

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

1 * factorial(0)

Factorial

factorial(1)

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

1 * factorial(0)

n = 0

```
if (n == 0)
    return 1;
return n * factorial(n - 1);
```

Factorial

factorial(1)

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

1 * factorial(0)

n = 0

```
if (n == 0)
    return 1;
return n * factorial(n - 1);
```

TRUE, return 1

Factorial

factorial(1)

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

1 * factorial(0)

n = 0

```
if (n == 0)
    return 1;
return n * factorial(n - 1);
```

TRUE, return 1

Factorial

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

1 * 1

factorial(1)

n = 0

```
if (n == 0)
    return 1;
return n * factorial(n - 1);
```

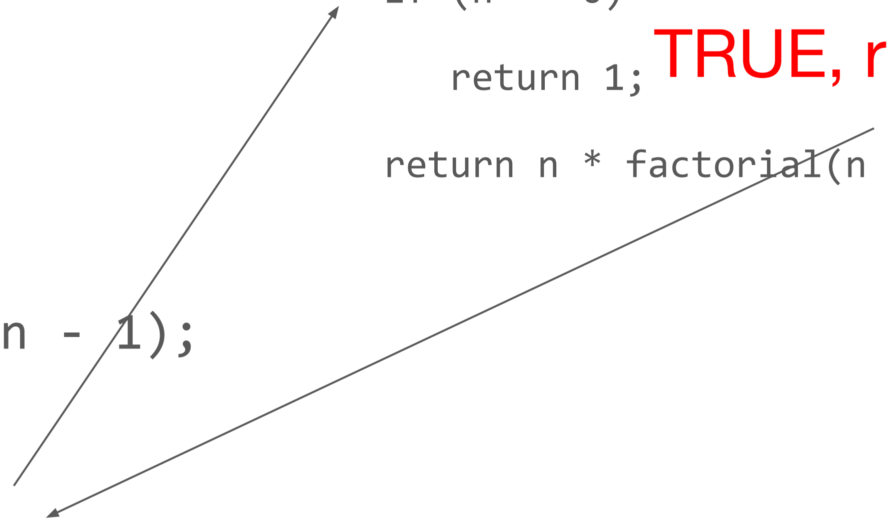
TRUE, return 1

Factorial

factorial(1)

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

1



n = 0

```
if (n == 0)
    return 1;
return n * factorial(n - 1);
```

TRUE, return 1

Fibonacci

Write a recursive function `fib` that computes the `n`th Fibonacci number. The 0th Fibonacci number is 0, the 1st Fibonacci number is 1, and every subsequent Fibonacci number is sum of the two preceding Fibonacci numbers.

Fibonacci

```
int fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}
```