CSCI 2132: Software Development

Functions, Abstraction & Recursion

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

Building Abstractions

Large projects are built in layers of abstraction!

Example: Computer

- We don't design a whole computer using individual transistors as building blocks!
- Computer = CPU, memory, graphics card, …
- CPU = registers, arithmetic/logic unit (ALU), command pipeline, …
- ALU = adders, …
- adder = half-adders, …
- half-adders = logic gates, …
- logic gates = transistors

Programming = building **software** abstractions

Functions

Functions = main abstraction method in programming

- Sequence of statements that can be used as a single functional unit
- Have arguments (input) and return values (output)

Function definition example:

```
int max(int a, int b) {
  int c;
  c = (a > b) ? a : b;
  return c;
}
```

Function Return Value

- `int` by default
  (Know to deal with legacy code, never use this in new code!)
- Arrays cannot be return values (but structs can!)

Example of calling a function:

- `printf("%d\n", max(a, b));`

Some standard functions:

- `printf`:
  - Return value = number of printed characters
- `scanf`:
  - Return value = number of read values or EOF if error occurs

Function Declaration vs Function Definition

Function declaration: Specifies only the argument and return types

```
int max(int a, int b);
```

```
int max(int, int);
```

(Since we only declare the function, the names of the arguments aren't needed, only their types.)

Function definition: Specifies what the function does

```
int max(int a, int b) {
  int c;
  c = (a > b) ? a : b;
  return c;
}
```

Why Declare a Function?

Evaluation order: For historic reasons, C compilers allow you to use a function or variable only after it has been declared.

Now consider:

```
int g(int);

int f(int x) {
  return g(x + 1);
}

int g(int x) {
  return (x > 10 ? x : f(x));
}
```

Why Declare a Function?

Information hiding:

max.h

```
#ifndef MAX_H
#define MAX_H

int max(int, int);

#endif MAX_H
```

max.c

```
#include “max.h”

int max(int a, int b) {
  return (a < b ? a : b);
}
```

Function Arguments

The terms **arguments** and **parameters** are often used interchangeably.

Strictly speaking, they are two different things.

Function arguments (or **actual parameters**):

- Values passed to a specific call of a function

```
max(a, 3 + (b - 1) / 2);
```

Function Arguments

The terms **arguments** and **parameters** are often used interchangeably.

Strictly speaking, they are two different things.

Function arguments (or **actual parameters**):

- Values passed to a specific call of a function

```
max(a, 3 + (b - 1) / 2);
```

Function parameters (or **formal parameters**):

- Names used to refer to the values passed to a function inside the function definition.

```
int max(int a, int b) {
  return (a < b) ? a : b;
}
```

Parameter Passing

Different programming languages allow different **parameter passing modes**:

- **By value:**
  - Modifications to the formal parameter inside the function body does not affect the caller.
- **By reference:**
  - Modifications to the formal parameter inside the function body modify the variable the caller provided as a function argument.
- **By sharing**, …

C passes all values by value.

Passing by reference and passing of array arguments accomplished by passing a pointer to the variable by value.

Example: A Swap Function, 1st Attempt

```
void swap(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}

int main() {
  int a = 4;
  int b = 5;
  swap(a, b);
  printf(“a = %d, b = %d\n”, a, b);
  return 0;
}
```

Example: A Swap Function, 2nd Attempt

```
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main() {
  int a = 4;
  int b = 5;
  swap(&a, &b);
  printf(“a = %d, b = %d\n”, a, b);
  return 0;
}
```

Passing Arrays to Functions

Remember: Arrays are just pointers to their first arguments, no size information.

Consequence: Most functions that work with arrays need an extra argument, the size of the array.

```
int max_array(int len, int a[]) {
 ...
}
```

or

```
int max_array(int len, int *a) {
 ...
}
```

Multidimensional Array Arguments

Multidimensional arrays can be passed as function arguments.

The compiler must know all dimensions, except possibly the first.

```
int f(int a[10][20]) { ... }
int g(float a[][50]) { ... }
```

For 1D arrays, `a[]` and `*a` are equivalent. For multidimensional arrays, `a[][]` and `*a[]` are not equivalent!

Multidimensional array:

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][0]	a[1][0]

Multidimensional Array Arguments

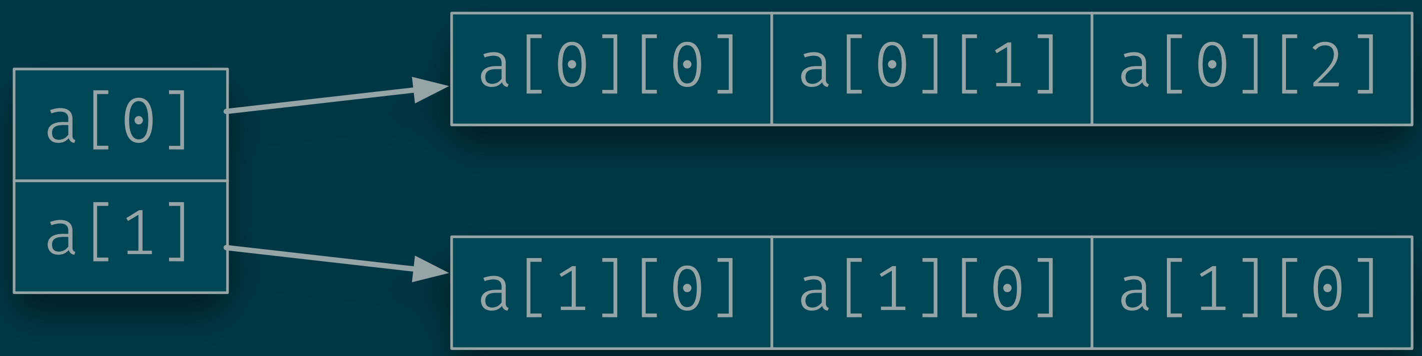
Multidimensional arrays can be passed as function arguments.

The compiler must know all dimensions, except possibly the first.

```
int f(int a[10][20]) { ... }
int g(float a[][50]) { ... }
```

For 1D arrays, `a[]` and `*a` are equivalent. For multidimensional arrays, `a[][]` and `*a[]` are not equivalent!

Array of pointers:



Variable-Length Multidimensional Arrays

Before C99:

- Pass a one-dimensional array (or pointer)
- Do index arithmetic explicitly

```
int matrix_multiply(int l, int m, int n,
                    float *a, float *b, float *c) {
  int i, j, k;
  for (i = 0; i < l; ++i)
    for (j = 0; j < n; ++j)
      c[i*n + j] = 0;
      for (k = 0; k < m; ++k) {
        c[i*n + j] += a[i*m + k] * a[k*n + j];
      }
}
```

Variable-Length Multidimensional Arrays

C99:

```
int matrix_multiply(int l, int m, int n,
                    float a[l][m], float b[m][n],
                    float c[l][n]) {
 for (int i = 0; i < l; ++i)
    for (int j = 0; j < n; ++j)
      c[i][j] = 0;
      for (int k = 0; k < m; ++k) {
        c[i][j] += a[i][k] * a[k][j];
      }
}
```

Variable-Length Multidimensional Arrays

C99:

```
int matrix_multiply(int l, int m, int n,
                    float a[][m], float b[][n],
                    float c[][n]) {
 for (int i = 0; i < l; ++i)
    for (int j = 0; j < n; ++j)
      c[i][j] = 0;
      for (int k = 0; k < m; ++k) {
        c[i][j] += a[i][k] * a[k][j];
      }
}
```

Variable-Length Multidimensional Arrays

Notes:

- The size must precede the array declaration:

```
int f(int n, int a[][n]) { ... }
```

  not

```
int f(int a[][n], int n) { ... }
```

- In declaration, the size parameters can be replaced by `*`:

```
int f(int, int a[][*]);
```

Pointer Arguments for Efficiency

Remember: All arguments are passed by value.

- This is costly for large structures
- **Solution:** Pass a pointer
- **But:** Now the function can modify the referenced data

Solution: `const` pointer

```
int f(const int *a) {
  // This is okay
  printf("%d", a[3]);
  // This is not
  a[4] = 0;
}
```

A Program’s Stack Supports Recursion

Every function call creates a **stack frame** or **activation record** on the stack:

- Return address
- Arguments
- Return value
- Local variables

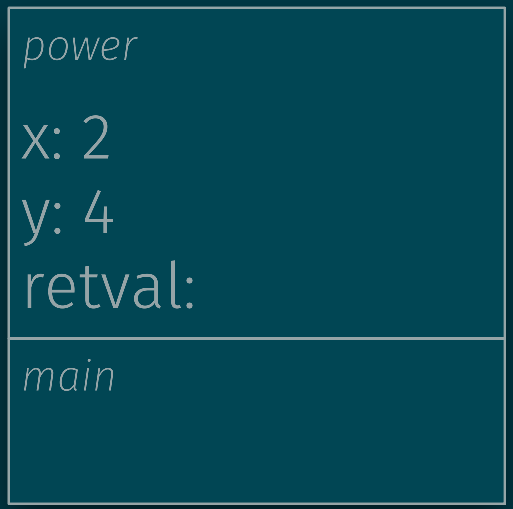
An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```

main

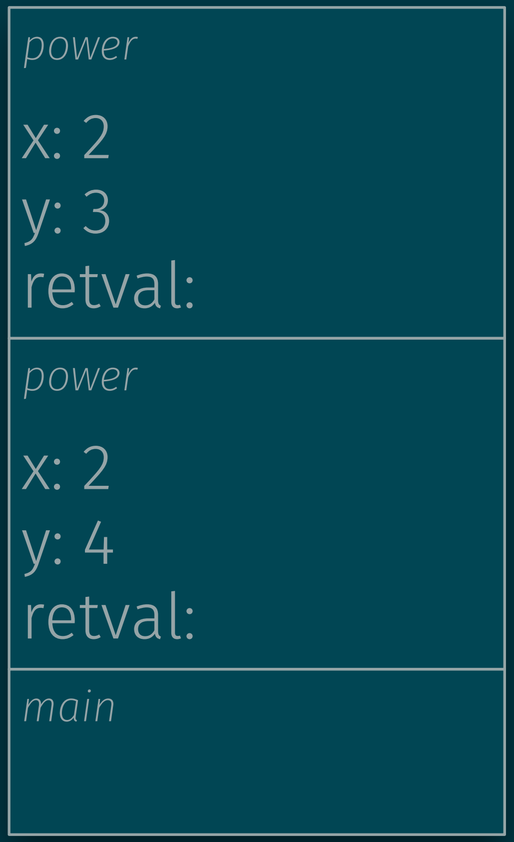
An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```



An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```



An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf("%d\n", power(2, 4));
}
```

power

x: 2
y: 2
retval:

power

x: 2
y: 3
retval:

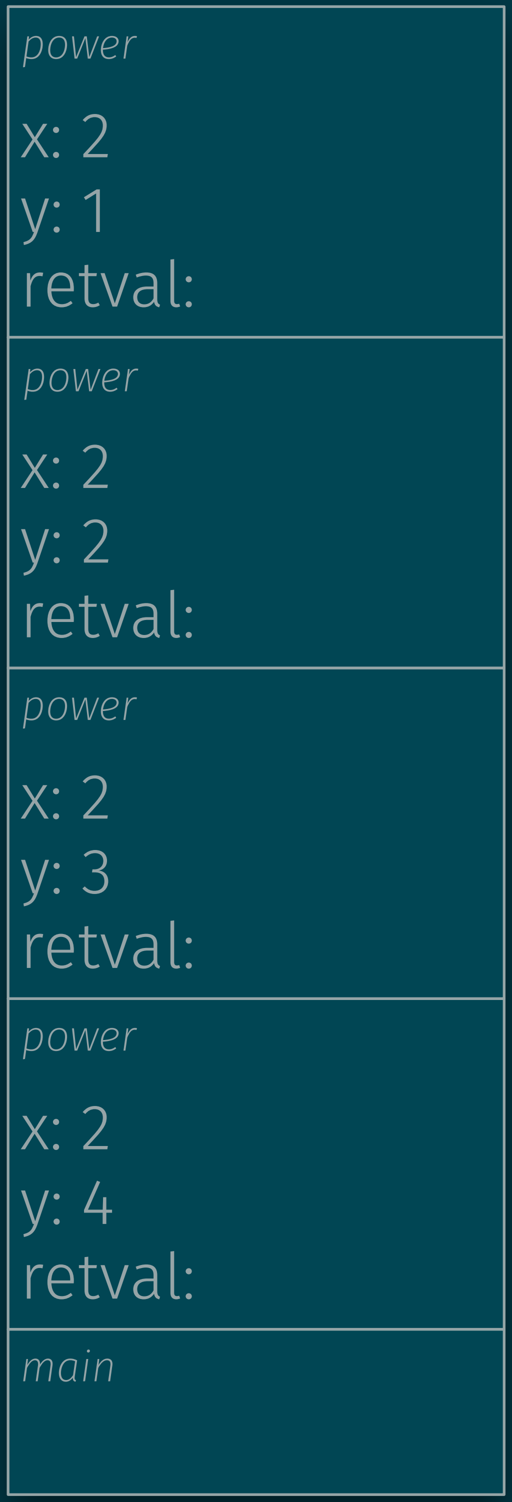
power

x: 2
y: 4
retval:

main

An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf("%d\n", power(2, 4));
}
```



An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```

power

x: 2
y: 0
retval:

power

x: 2
y: 1
retval:

power

x: 2
y: 2
retval:

power

x: 2
y: 3
retval:

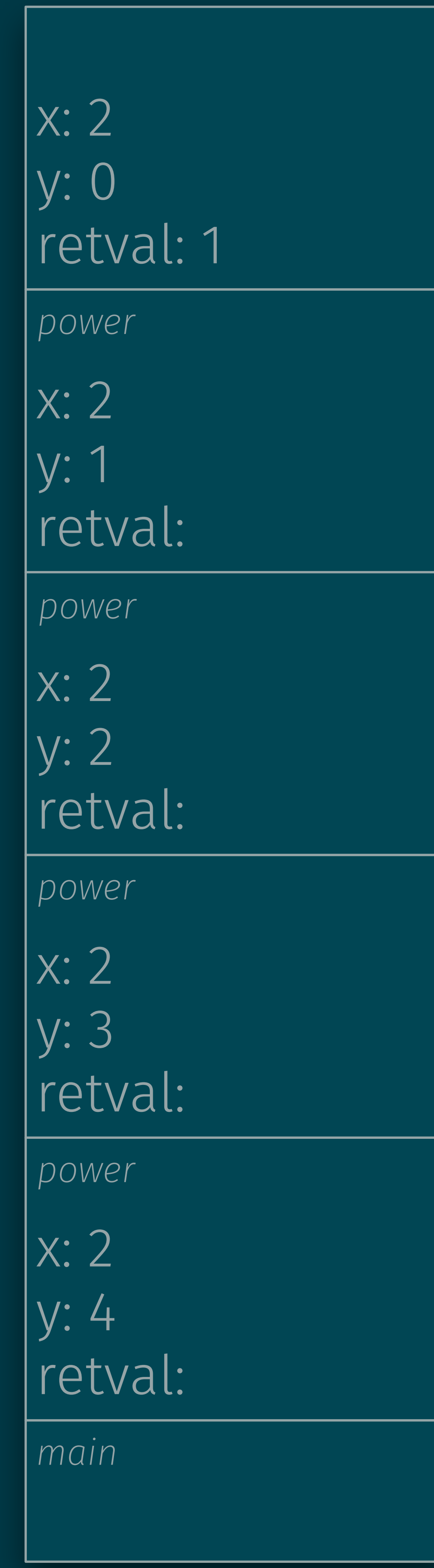
power

x: 2
y: 4
retval:

main

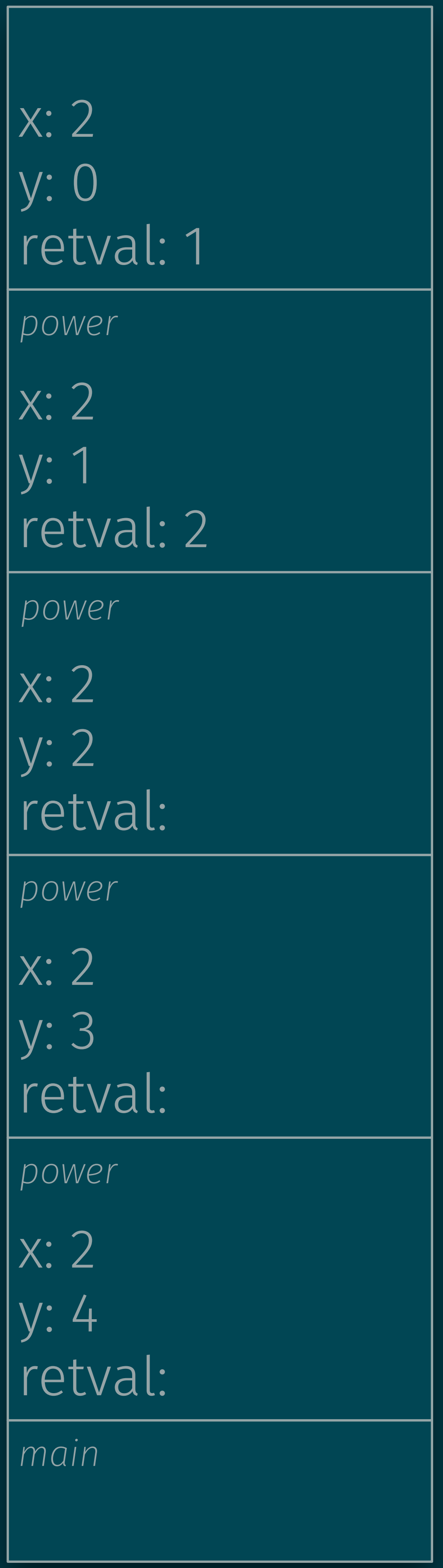
An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```



An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```



An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```

power x: 2 y: 1 retval: 2
power x: 2 y: 2 retval:
power x: 2 y: 3 retval:
power x: 2 y: 4 retval:
main

An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```

power

x: 2
y: 1
retval: 2

power

x: 2
y: 2
retval: 4

power

x: 2
y: 3
retval:

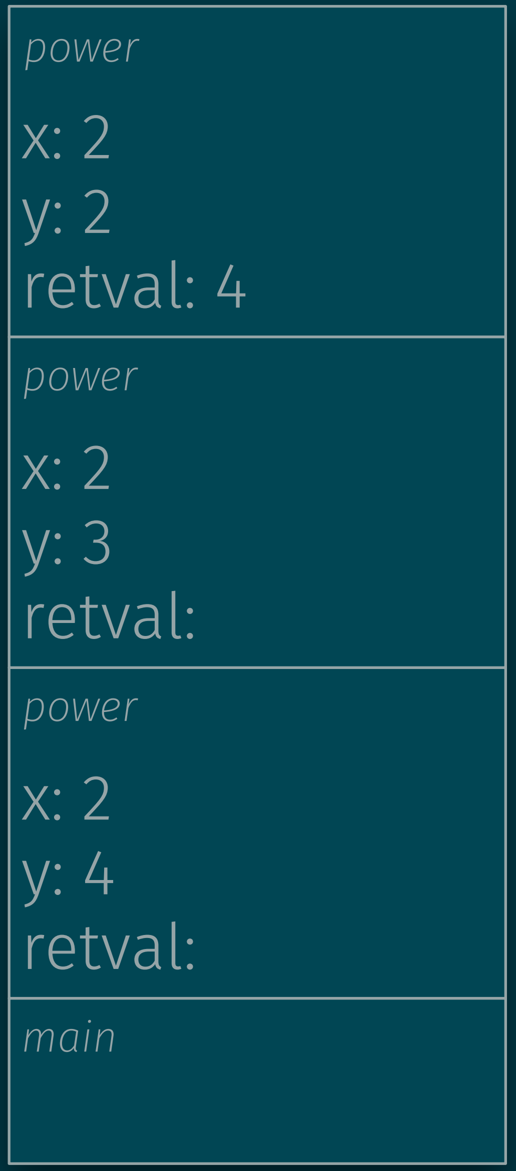
power

x: 2
y: 4
retval:

main

An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf("%d\n", power(2, 4));
}
```



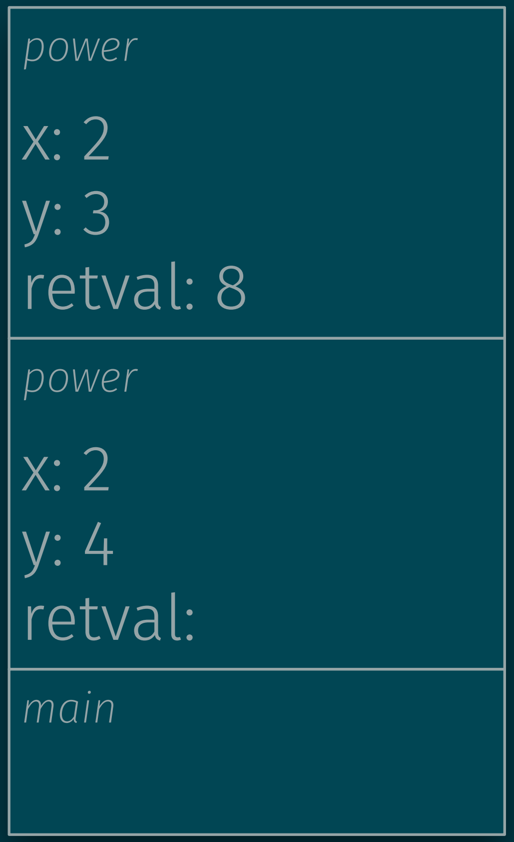
An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```

power x: 2 y: 2 retval: 4
power x: 2 y: 3 retval: 8
power x: 2 y: 4 retval:
main

An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf("%d\n", power(2, 4));
}
```



An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```

power

x: 2
y: 3
retval: 8

power

x: 2
y: 4
retval: 16

main

An Example

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```

power

x: 2
y: 4
retval: 16

main

An Example

```c
int power(int x, int y) {
  if (y == 0) {
    return 1;
  }
  return x * power(x, n-1);
}

int main() {
  printf(“%d\n”, power(2, 4));
}
```

main

Lexical Scopes

Local variables, function arguments are visible (can be accessed) only inside the function.

"Normal" local variables and function arguments exist only while the function call is active.

`static` local variables

- Outlive function call.
- Have the same memory location in each function call.
- Stored in the DATA segment of the process's memory.

```
char *gentmp() {
  static char tmp[16];
  static int i = 0;
  sprintf(tmp, “tmp%d.txt”, i);
  return tmp;
}
```

Recursion

Recursive functions call themselves.

Mutually recursive functions call each other.

Each function call has its own stack frame (local variables, …).

Two ways to repeat things:

- Iteration (or tail-recursion)
- Recursion (a function can call itself more than once)

```
int f( ... ) {
  ...
  if ( ... ) {
    f( ... );
  }
  ...
}
```

```
int f( ... ) {
  ...
  if ( ... ) {
    g( ... );
  }
  ...
}

int g( ... ) {
  ...
  f( ... );
  ...
}
```

Example: Computing Fibonacci Numbers

Note: There is an iterative way to do this in linear time.

The following recursive solution takes exponential time but matches the formula.

$$F_n = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

```
int fib(int n) {
  if (n > 1) {
    return fib(n-1) + fib(n-2);
  } else {
    return 1;
  }
}
```

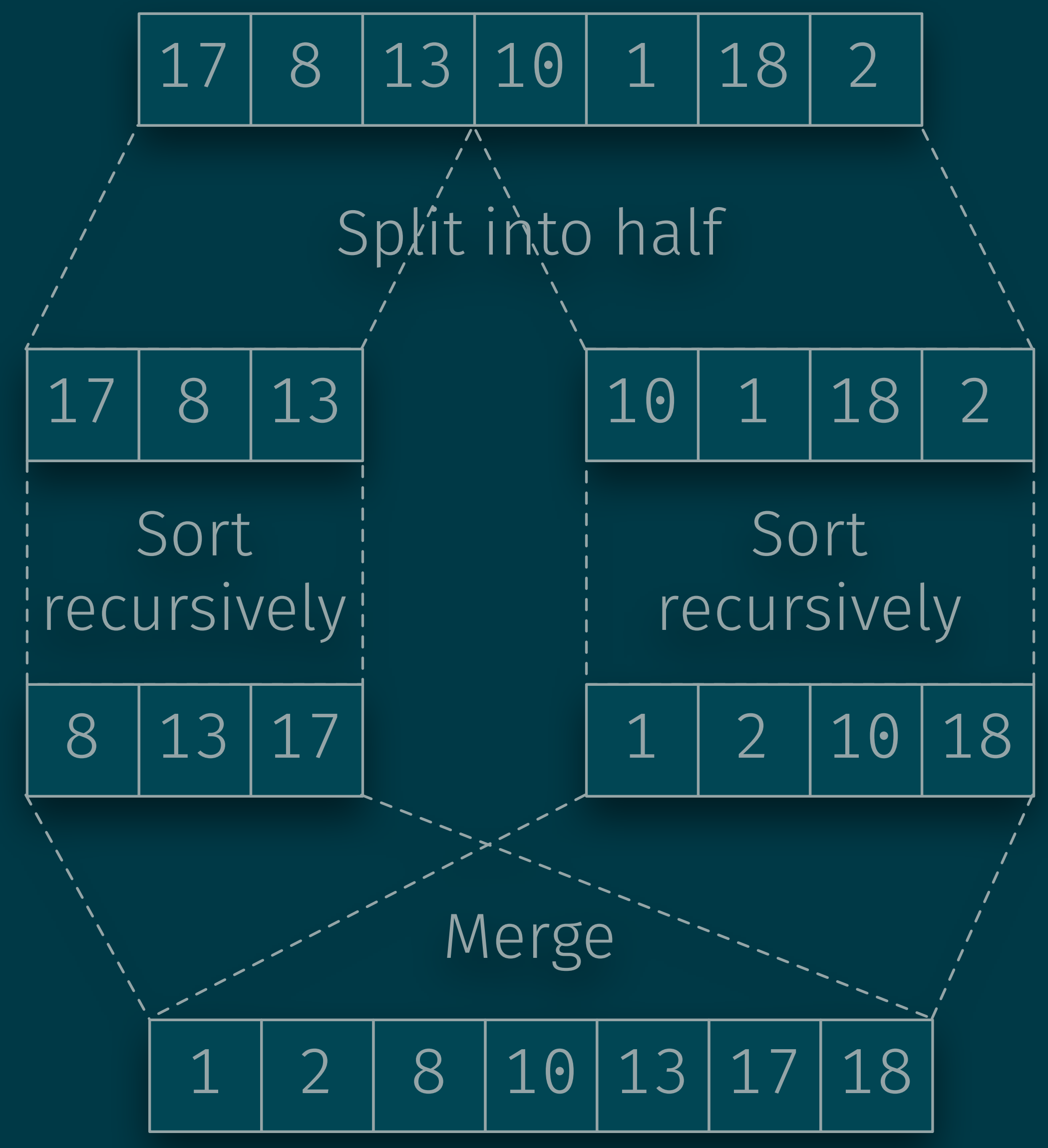
Merge Sort

Sorting by Forming Longer Sorted Sequences

An inductive approach (recursion can often be viewed as induction):

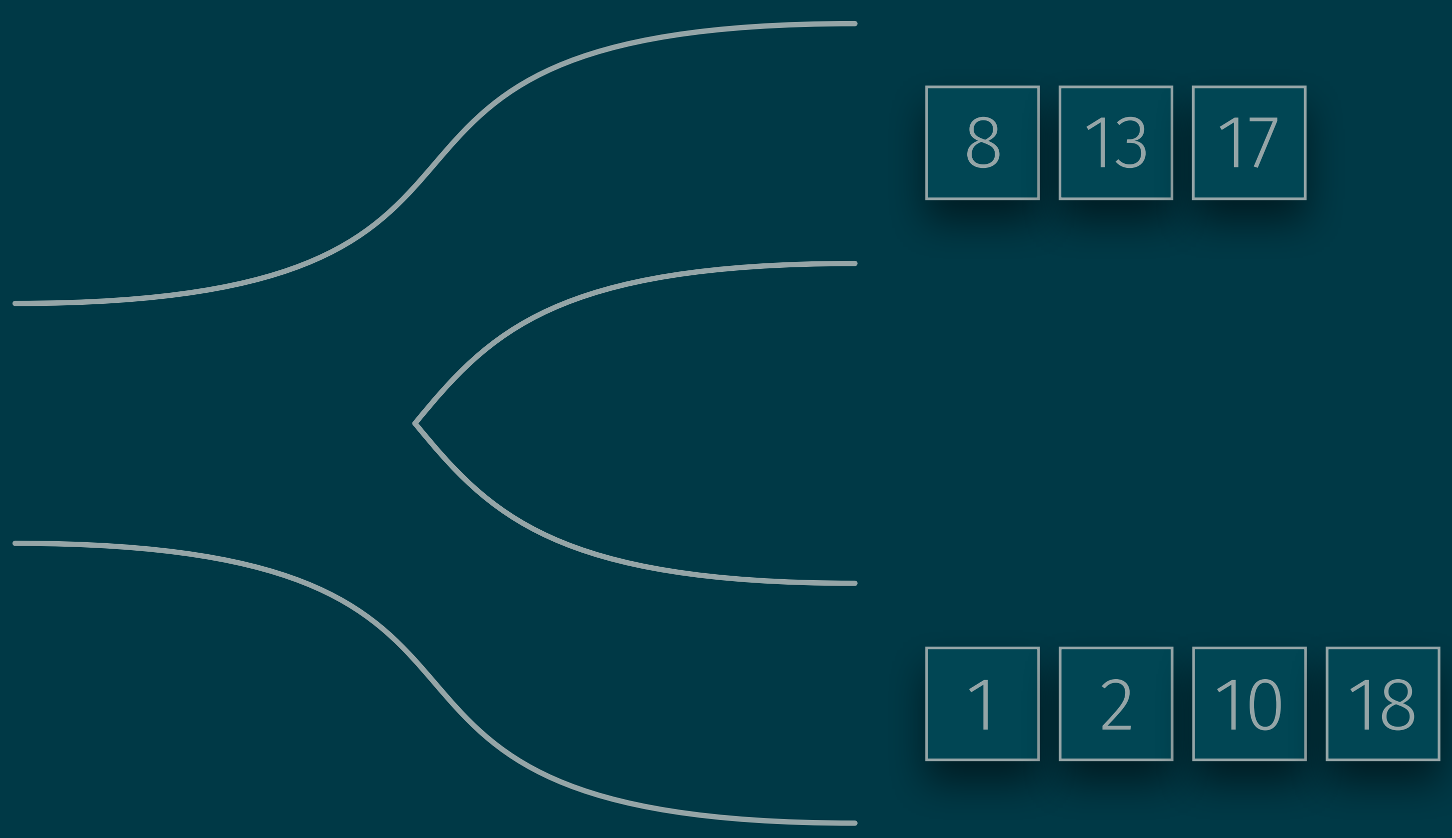
- If $|A| < 2$, then A is sorted
- Otherwise, we
  - Inductively (recursively) sort the left and right halves
  - Merge the resulting two sorted lists

Key idea: Reduce sorting to the easier problem of merging two sorted sequences.

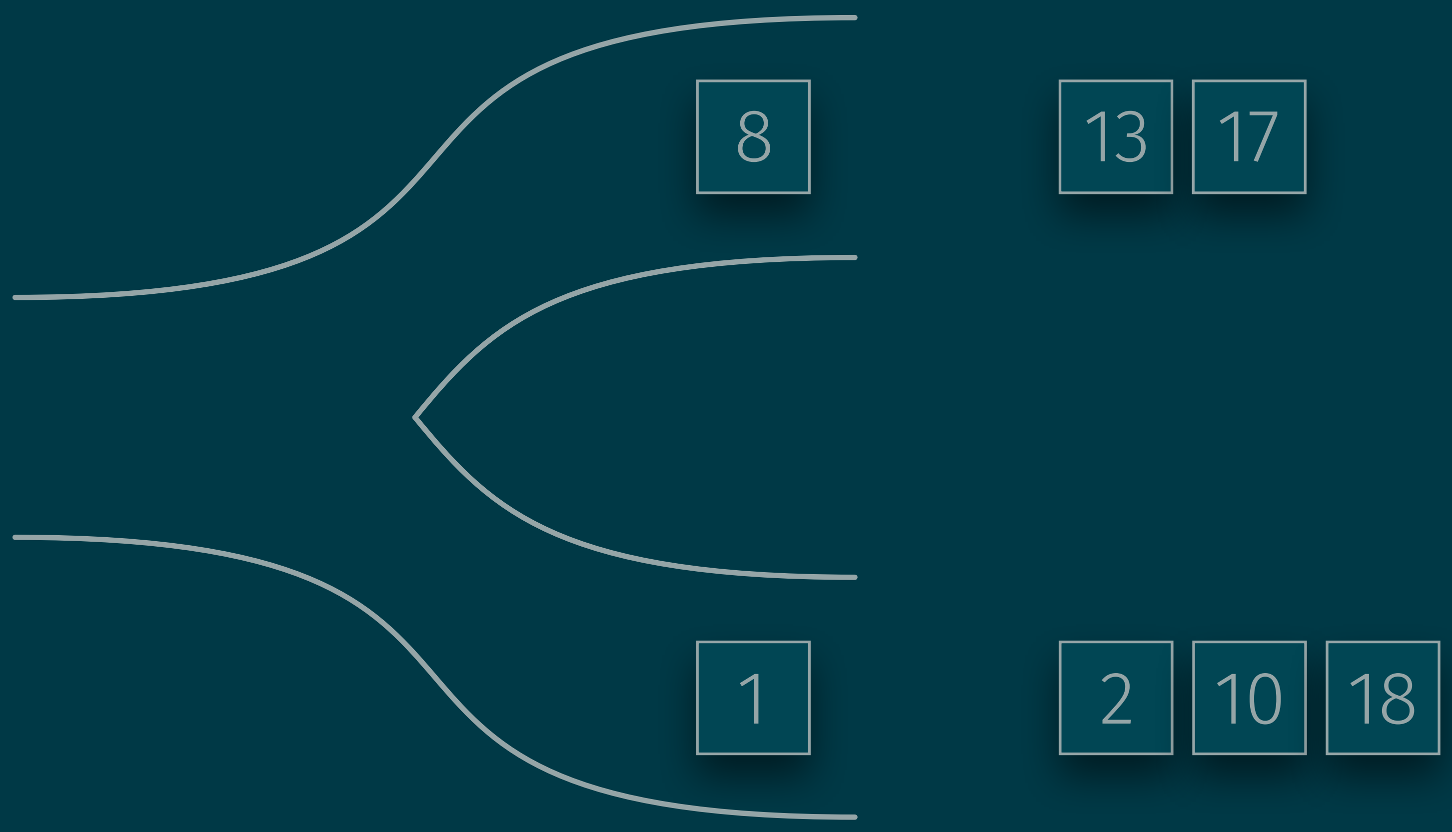



```
void mergesort(int n, int *a) {
  if (n > 1) {
    int m = n / 2;
    mergesort(a, m);
    mergesort(a + m, n - m);
    merge(a, m, n);
  }
}
```

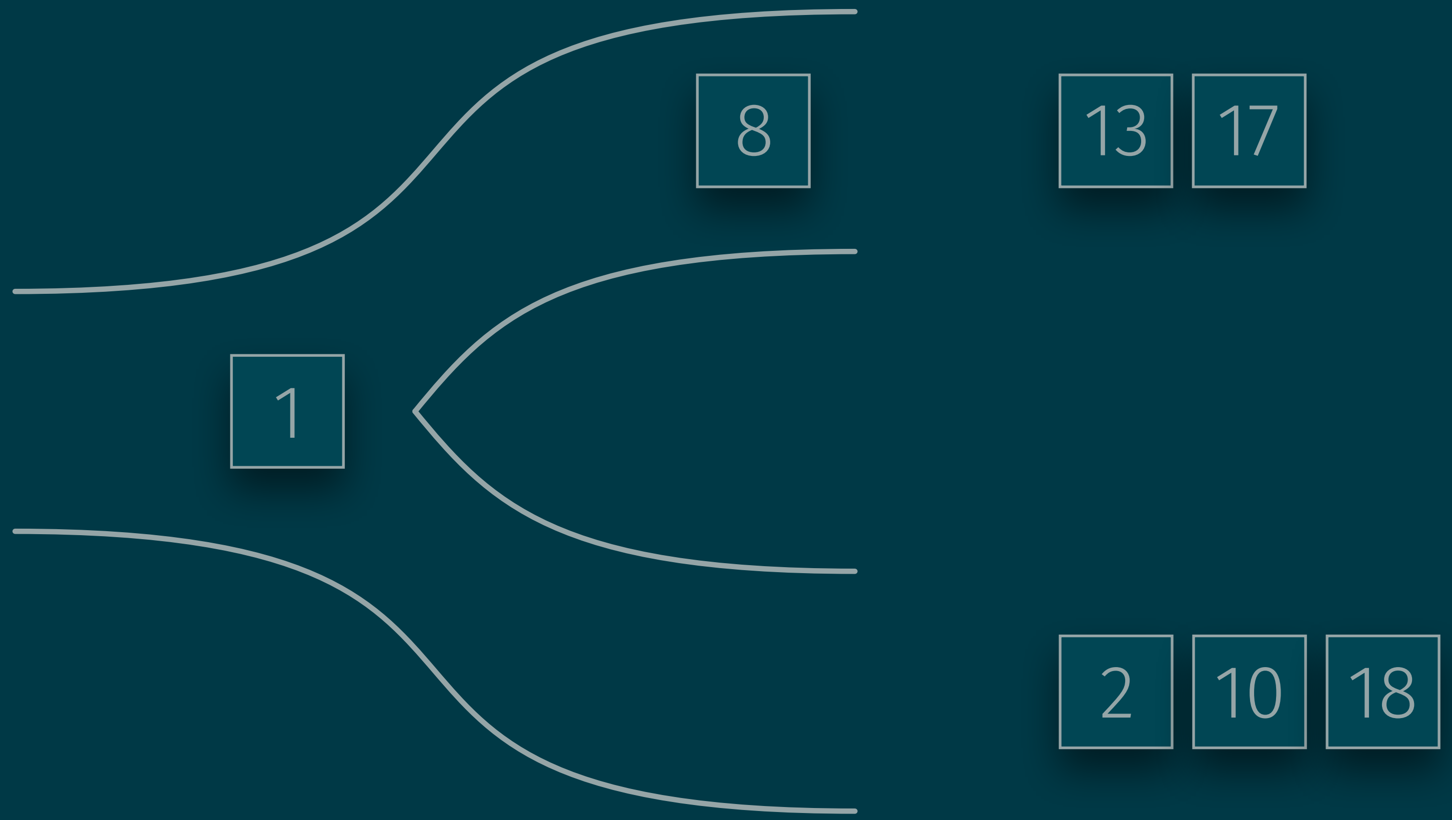
Merging



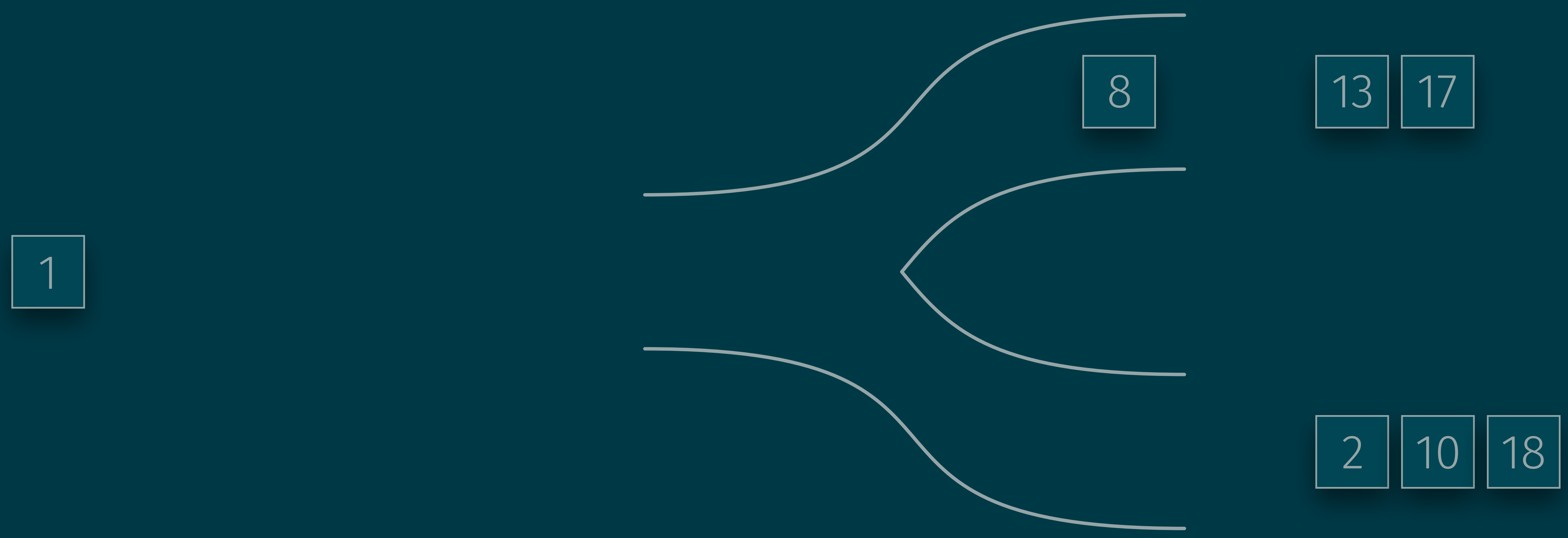
Merging



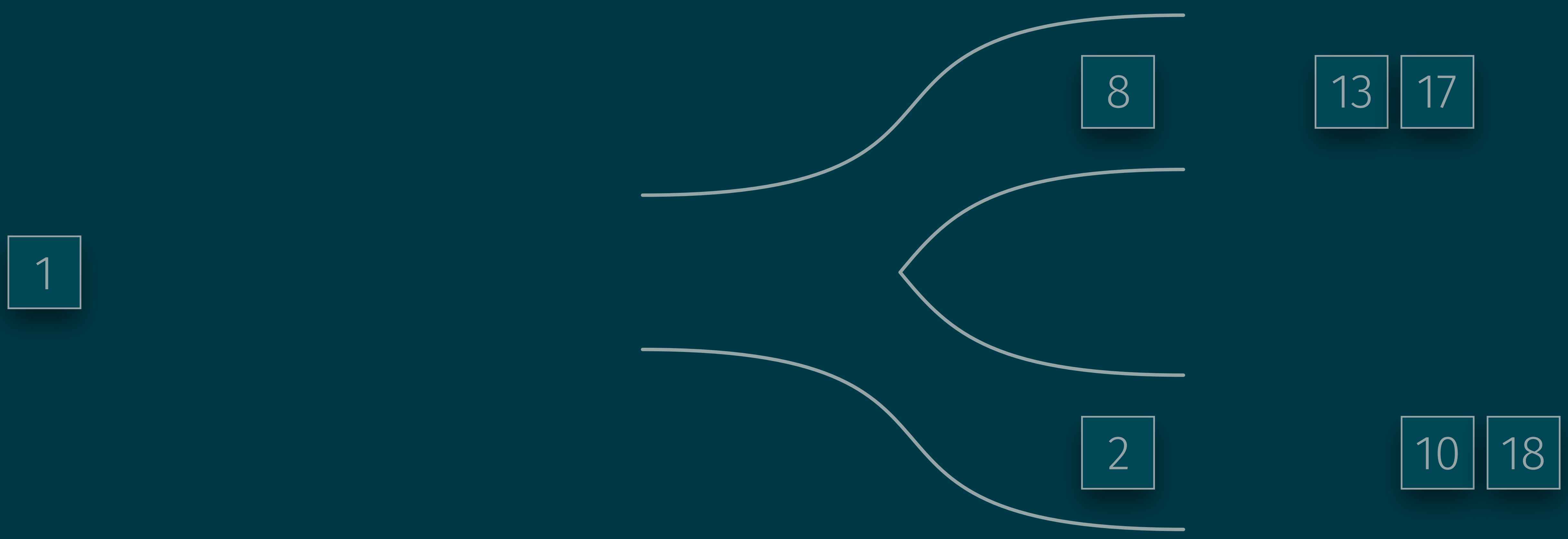
Merging



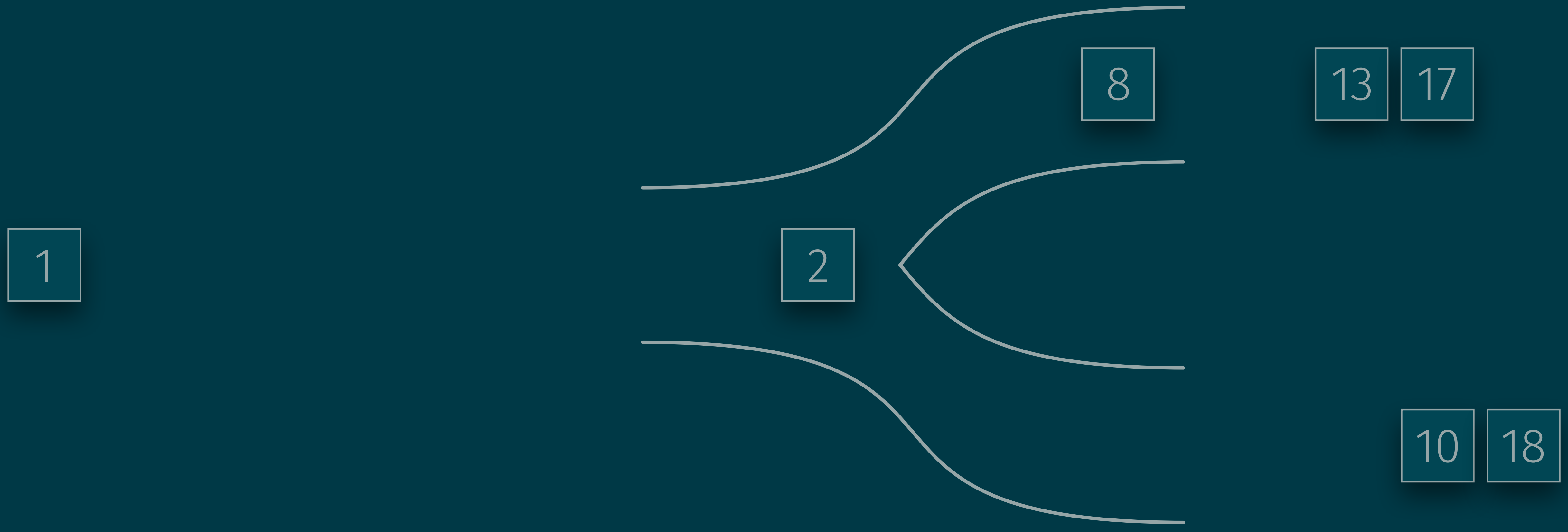
Merging



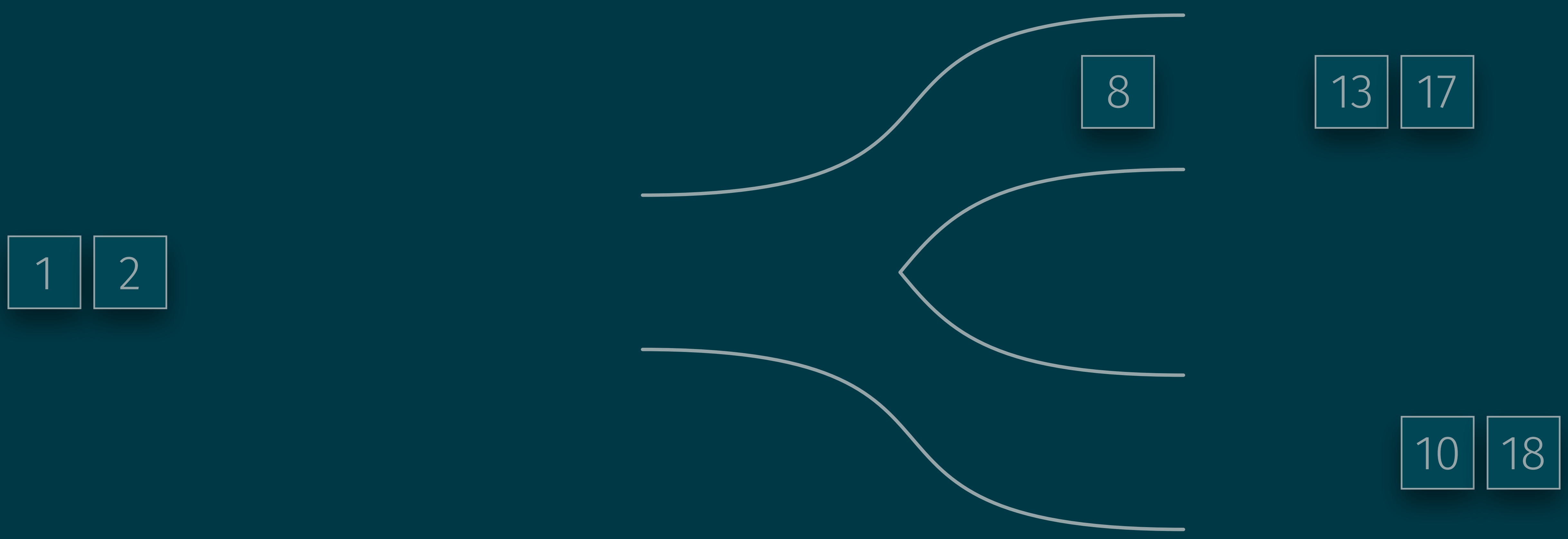
Merging



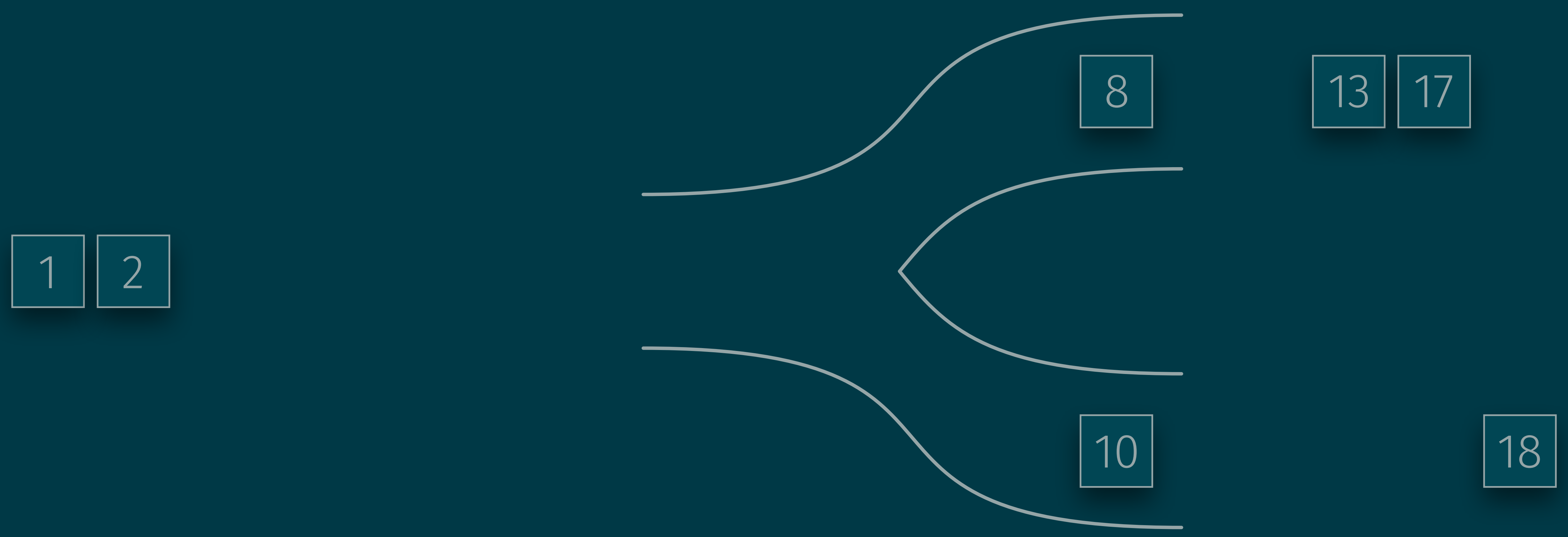
Merging



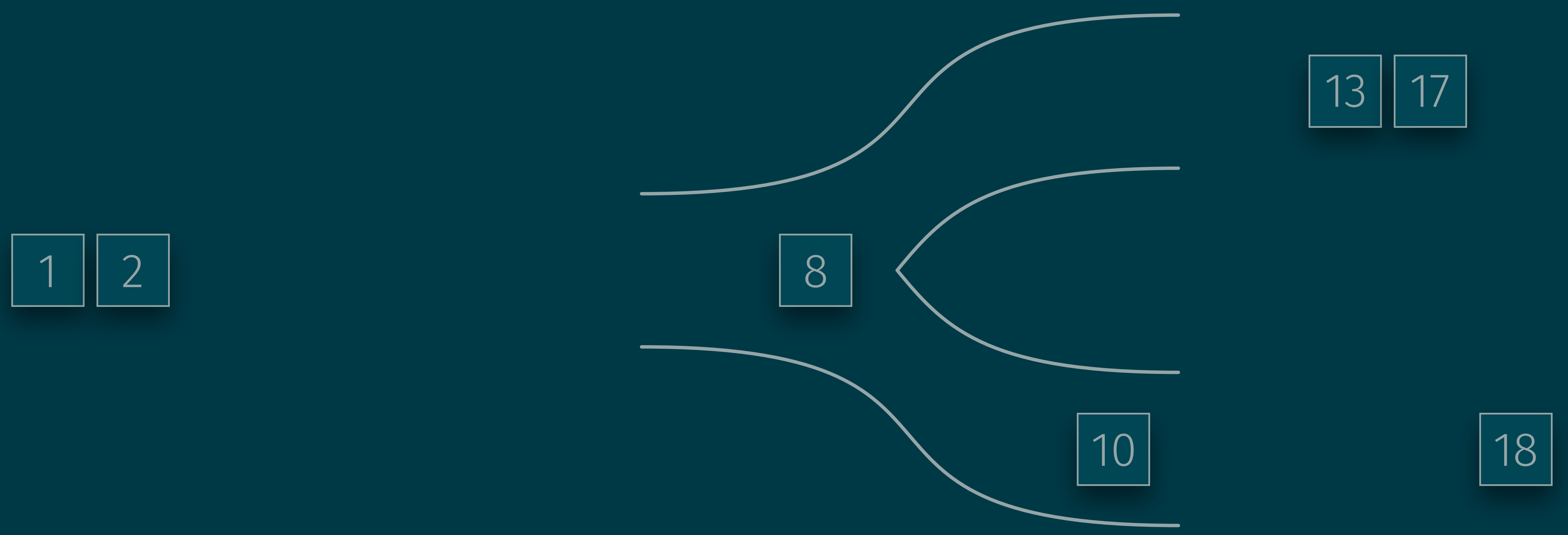
Merging



Merging

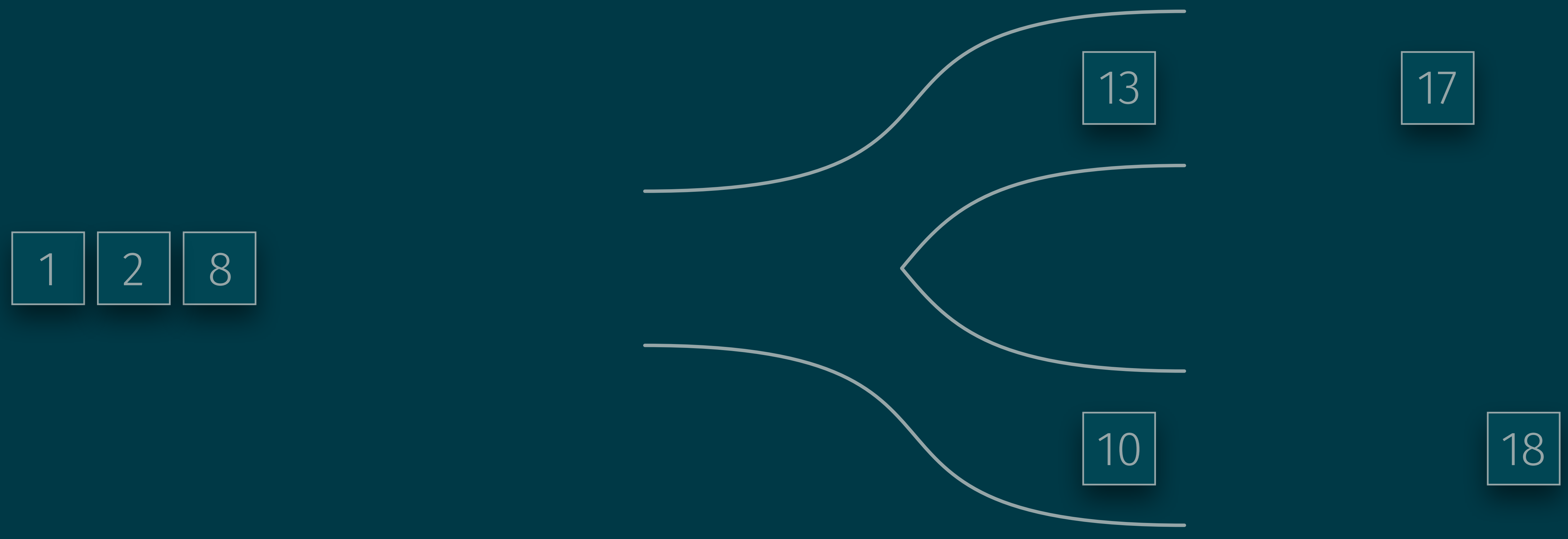


Merging

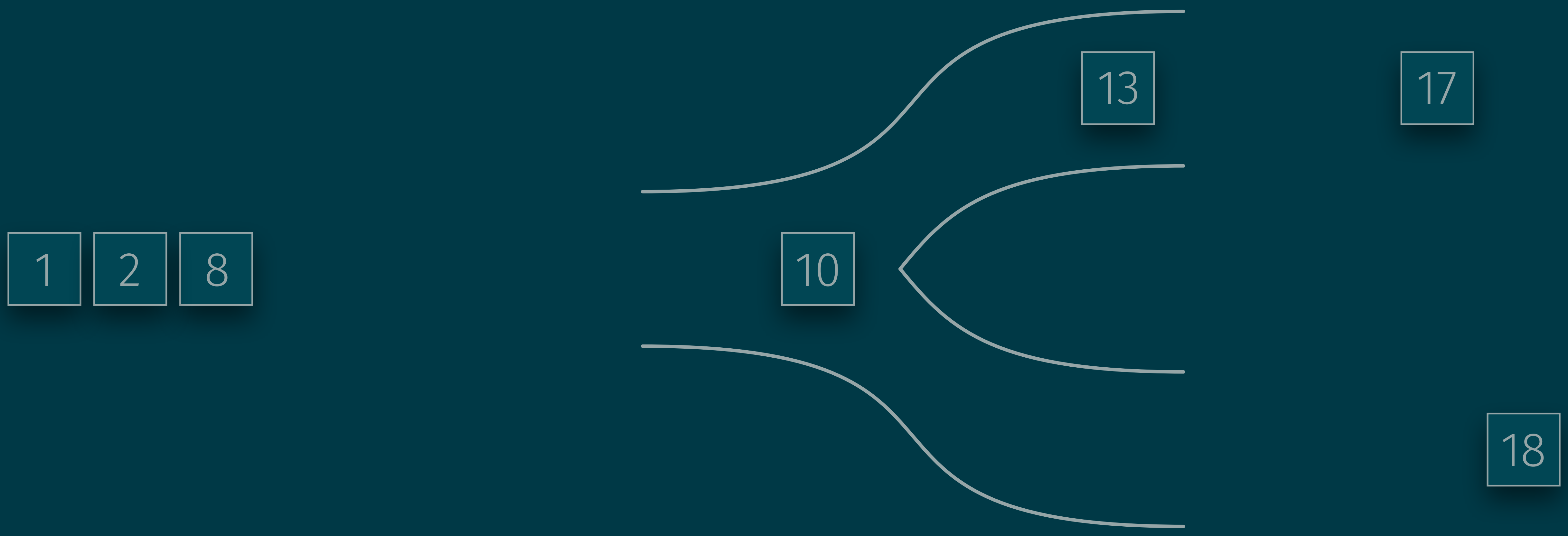


Merging

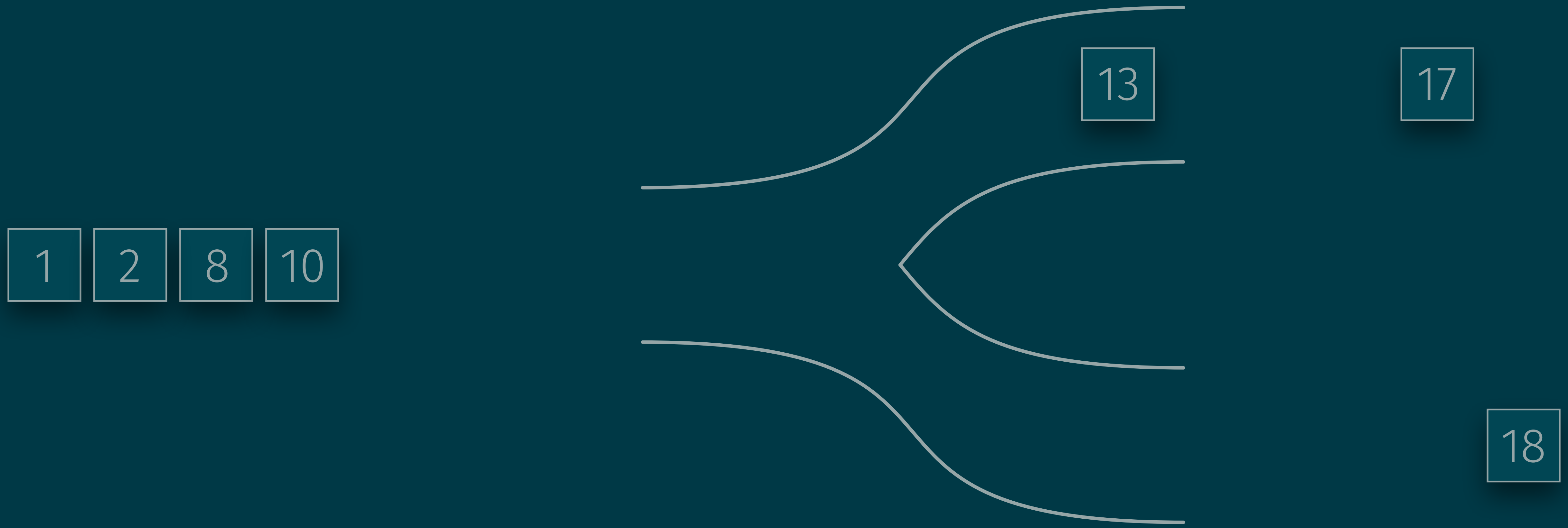
Merging



Merging



Merging



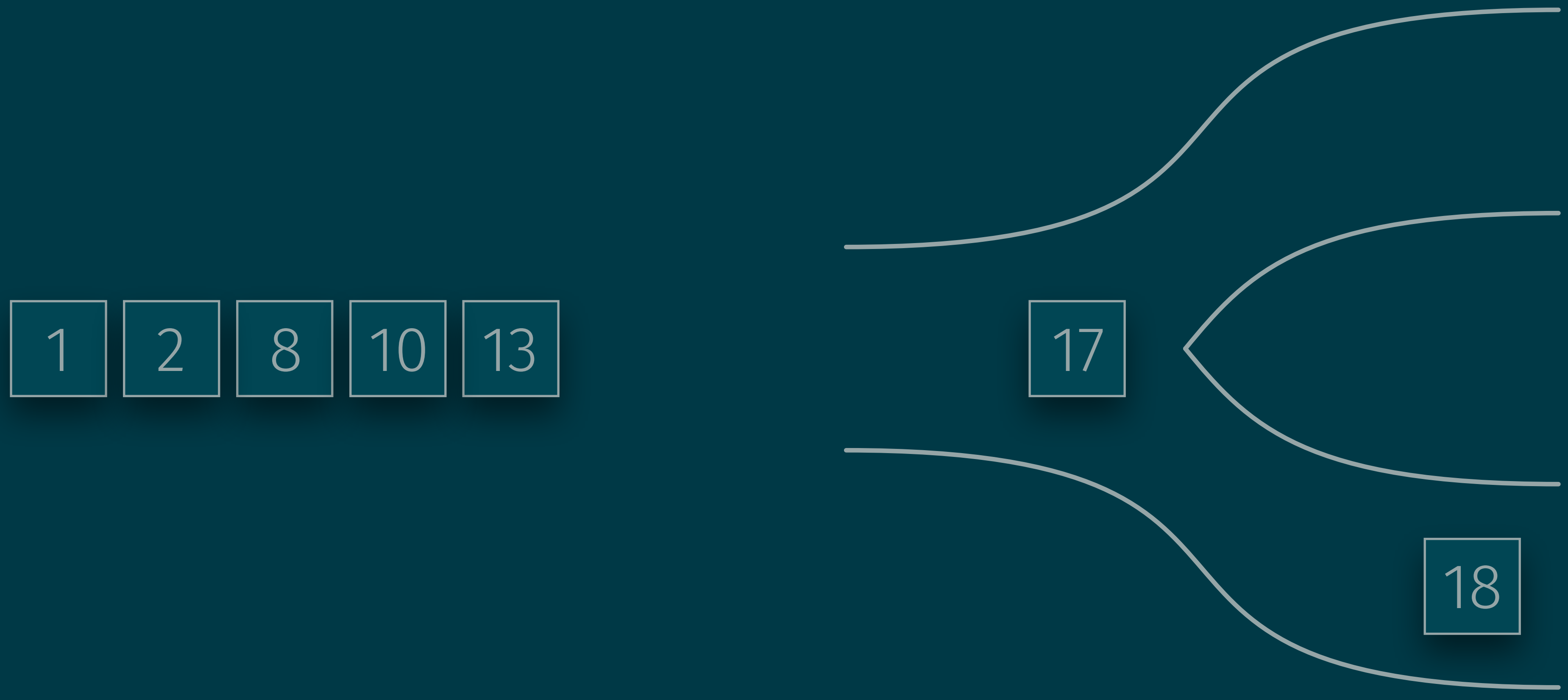
Merging

Merging

Merging

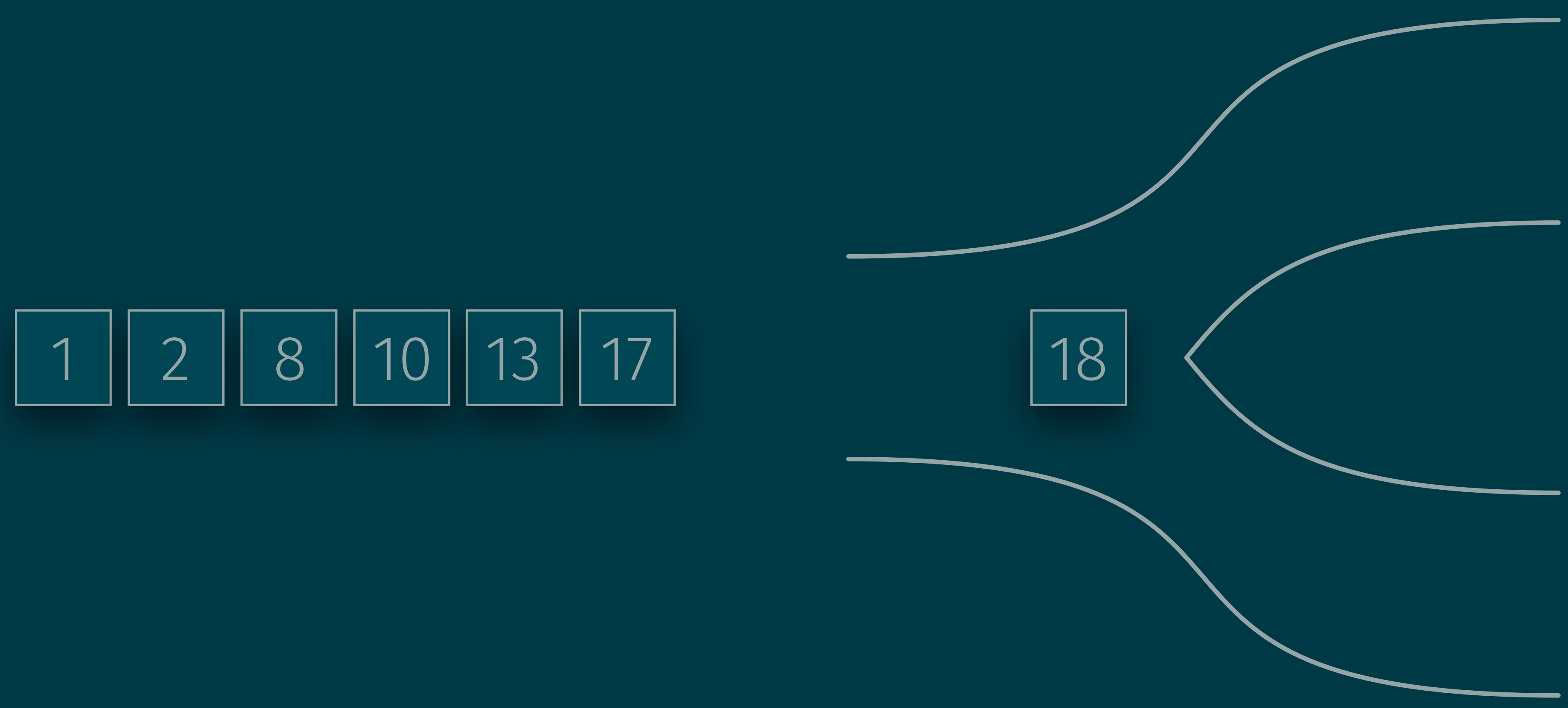
Merging

Merging



Merging

Merging



Merging

Merging

```
void merge(int *a, int m, int n) {
  int tmp[n], i, j, k;
  memcpy(tmp, a, n * sizeof(int));
  for (i = 0, j = m, k = 0; i < m && j < n; ++k) {
    if (tmp[j] < tmp[i]) {
      a[k] = tmp[j++];
    } else {
      a[k] = tmp[i++];
    }
  }
  while (i < m) {
    a[k++] = tmp[i++];
  }
  while (j < n) {
    a[k++] = tmp[j++];
  }
}
```

Merge Sort for Arbitrary Element Types

```
void mergesort(void *a, int elem_size, int n,
               int (*cmp)(const void *, const void *)) {
  if (n > 1) {
    int m = n / 2;
    mergesort(a, elem_size, m, cmp);
    mergesort(a + m * elem_size, n - m, cmp);
    merge(a, elem_size, m, n, cmp);
  }
}
```

Merging with Arbitrary Element Types

```
void merge(void *a, int elsz, int m, int n,
           int (*cmp)(const void *, const void *)) {
  char tmp[n * elsz];
  int i, j, k;
  memcpy(tmp, a, n * elsz);
  for (i = 0, j = m, k = 0; i < m && j < n; ++k) {
    if (cmp(tmp + j * elem_size, tmp + i * elem_size) < 0) {
      memcpy(a + k * elsz, tmp + (j++) * elsz, elsz);
    } else {
      memcpy(a + k * elsz, tmp + (i++) * elsz, elsz);
    }
  }
  while (i < m) {
    memcpy(a + (k++) * elsz, tmp + (i++) * elsz, elsz);
  }
  while (j < n) {
    memcpy(a + (k++) * elsz, amp + (j++) * elsz, elsz);
  }
}
```

Running Time of Merge Sort

Merging takes linear time, constant time per element.

How many merge steps is each element involved in?
(Assume $n = 2^k$)

- Merged sizes: 2, 4, 8, 16, ..., n
- lg n merge steps

Total running time ~ $n \lg n$

Compare: Insertion Sort takes ~ n^2 time

Practically faster sorting algorithm with $n \lg n$ running time in expectation:
Quick Sort