In [6]:

```
# setup
from IPython.core.display import display,HTML
display(HTML('<style>.prompt{width: 0px; min-width: 0px; visibility: collapse}</style>'))
display(HTML(open('../rise.css').read()))

# imports
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set(style="whitegrid", font_scale=1.5, rc={'figure.figsize':(12, 6)})
```

CMPS 2200

Introduction to Algorithms

Divide and Conquer

Today's agenda:

- Divide-and-Conquer Framework
- Sequence Scan

We've seen a few divide-and-conquer algorithms already, so let's look at the high-level approach. For a problem A and instance $\mathcal{I}_A$:

- **Base Case**: If $\mathcal{I}_A$ is small, solve directly.
- **Inductive Step**:
  - **Divide** $\mathcal{I}_A$ into smaller instances.
  - **Recursively solve** smaller instances.
  - **Combine** solutions

As we'll see, each algorithmic paradigm has high-level strategies to i) *prove correctness* and ii) *determine work/span*.

How do we prove the correctness of a divide-and-conquer algorithm?

Induction -- why and how?

Induction provides a natural framework for divide-and-conquer algorithms.

The **base case** of the induction requires us to prove that the algorithm works for the base case.

For the **induction step**, we use the inductive hypothesis that the solutions to the smaller instances are correct. Then, we must prove that the combine step correctly produces the desired solution.

What about determining work/span?

We've seen that recurrences can capture the behavior of divide-and-conquer algorithms - they simply capture the cost of recursively solving smaller instances and then combining the solutions.

The general form of the work is:

$$W(n) = W_{\text{divide}}(n) + \sum_{i=1}^{k} W(n_i) + W_{\text{combine}}(n) + 1$$

The general form for the span is:

$$S(n) = S_{\text{divide}}(n) + \max_{i=1}^{k} S(n_i) + S_{\text{combine}}(n) + 1$$

Let's look at how Merge Sort fits into this framework.

$$
\begin{array}{l}
mergeSort\, a = \\
\quad \texttt{if } |a| \leq 1 \texttt{ then} \\
\quad\quad a \\
\quad \texttt{else} \\
\quad\quad \texttt{let} \\
\quad\quad\quad (l, r) = splitMid\, a \\
\quad\quad\quad (l', r') = (mergeSort\, l \;||\; mergeSort\, r) \\
\quad\quad \texttt{in} \\
\quad\quad\quad merge(l', r') \\
\quad\quad \texttt{end}
\end{array}
$$

For the correctness, we need to show that Merge Sort truly sorts the list. Let's perform induction:

- **Base case**: We correctly sort a singleton list.
- **Induction Step**: We assume that we can correctly sort the two halves of the list (by the inductive hypothesis). The final step is to prove that the merge step works correctly to combine two sorted lists into one sorted list (which we've shown previously).

For the running time, recall that we characterized the work/span of Merge Sort as:

$$W(n) = 2W(n/2) + O(n)$$

and

$$S(n) = S(n/2) + O(\log n)$$

This fits into the framework above: the divide step takes $O(1)$ time, there are 2 concurrent recursive calls, and merging takes $O(n)$ work and $O(\log n)$ span.

Now let's look at `scan`. We developed a contraction-based algorithm for this problem, but let's look at a divide-and-conquer strategy.

Remember that we looked at taking prefix sums for intuition. Let's reuse that example:

$\texttt{prefix\_sum}([2, 1, 3, 2, 2, 5, 4, 1]) \to ([0, 2, 3, 6, 8, 10, 15, 19], 20)$

Now, instead of contracting pairs of entries what if we just split the list and recursively compute prefix sums?

We'd get results (b, b') and (c, c') where:

$(b, b') = (\langle 0, 2, 3, 6 \rangle, 8)$

$(c, c') = (\langle 0, 2, 7, 11 \rangle, 12)$

Now, it's easy to see that b already gives us half the solution - how do we get the result by combining solutions?

To compute prefix sums, all we have to do is to add the sum of the first half to all of the elements of b and to b'. We can generalize this approach to get:

$$
\begin{array}{l}
scanDC\ f\ id\ a = \\
\quad \texttt{if}\ |a| = 0\ \texttt{then} \\
\quad\quad (\langle\ \rangle, id) \\
\quad \texttt{else if}\ |a| = 1\ \texttt{then} \\
\quad\quad (\langle\ id \rangle, a[0]) \\
\quad \texttt{else} \\
\quad\quad \texttt{let} \\
\quad\quad\quad (b, c) = splitMid\ a \\
\quad\quad\quad ((l, b'), (r, c')) = (scanDC\ f\ id\ b\ ||\ \ scanDC\ f\ id\ c) \\
\quad\quad\quad r' = \langle\ f(b', x) : x \in r\ \rangle \\
\quad\quad \texttt{in} \\
\quad\quad\quad (append\ (l, r'), f(b', c')) \\
\quad\quad \texttt{end}
\end{array}
$$

How do we prove the correctness of this algorithm?

- **Base Case**: For empty and singleton lists, we compute the correct result.
- **Induction Step**: For the induction hypothesis, we assume that we correctly compute the scan results for the two halves of the list. For the combine step, we compute $f(b', x)$ for all $x \in r$. Why is this the correct result for the right half of the list?

This is because $b' = f(l[n/2], a[n/2])$ and $l[n/2]$ is the correct scan result for the left half of the list. Consider the first element of r', it is $f(b', id) = b' = f(l[n/2], a[n/2])$. This is the correct scan result for a at position $n/2 + 1$.

We can similarly show that each successive element in r' is the correct scan result for a in the right half of the solution. Finally, since f is associative we can see that $f(b', c')$ is the correct "sum" result as well.

Assuming that $f(n)$ can be computed in constant time, we get the following recurrences for work and span:

$$W(n) = 2W(n/2) + O(n)$$

and

$$S(n) = S(n/2) + O(1)$$

Thus the work is $O(n \log n)$ and the span is $O(\log n)$.

Is this algorithm work-efficient? Why or why not?

Do you find `scanDC` more or less intuitive than the version using contract?

In [8]:

```
def scanDC(f, id_, a):
    space = len(a) * '  ' # for printing
    print(space, 'a=', a)

    if len(a) == 0:
        return ([], id_)
    elif len(a) == 1:
        return ([id_], a[0])
    else:
        b = a[:len(a)//2]
        c = a[len(a)//2:]
        left, L = scanDC(f, id_, b)
        right, R = scanDC(f, id_, c)
        updated_right = [f(L, x) for x in right]
        return left + updated_right, f(L, R)

def add(x,y):
    return x + y

scanDC(add, 0, [2,1,3,2,2,5,4,1])
```

```
                  a= [2, 1, 3, 2, 2, 5, 4, 1]
          a= [2, 1, 3, 2]
      a= [2, 1]
    a= [2]
    a= [1]
      a= [3, 2]
    a= [3]
    a= [2]
          a= [2, 5, 4, 1]
      a= [2, 5]
    a= [2]
    a= [5]
      a= [4, 1]
    a= [4]
    a= [1]
```

Out[8]:
```
([0, 2, 3, 6, 8, 10, 15, 19], 20)
```

$$
\begin{array}{l}
ternaryScanDC\ f\ id\ a = \\
\quad \textbf{if } |a| = 0 \textbf{ then} \\
\quad\quad (\langle\,\rangle, id) \\
\quad \textbf{else if } |a| = 1 \textbf{ then} \\
\quad\quad (\langle\, id \rangle, a[0]) \\
\quad \textbf{else} \\
\quad\quad \textbf{let} \\
\quad\quad\quad b = a[0 \cdots |a|/3] \\
\quad\quad\quad c = a[|a|/3 \cdots 2|a|/3] \\
\quad\quad\quad d = a[2|a|/3 \cdots] \\
\\
\quad\quad\quad ((l, b'), (m, d'), (r, c') = (scanDC\ f\ id\ b \;||\; scanDC\ f\ id\ c \;||\; scanDC\ f\ id\ d) \\
\quad\quad\quad r' = \langle\, f(b', x) : x \in m \,\rangle \\
\quad\quad \textbf{in} \\
\quad\quad\quad (append\ (l, r'), f(b', c')) \\
\quad\quad \textbf{end}
\end{array}
$$