In [4]:

```
# 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

MCSS and Euclidean TSP

Today's agenda:

- Divide-and-Conquer with `reduce`
- Maximum Contiguous Subsequence Sum
- Euclidean TSP

Recall that we gave a divide-and-conquer algorithm for `reduce`:

$$reduce\ f\ id\ a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f(reduce\ f\ id\ (a[0 \ldots \lfloor \frac{|a|}{2} \rfloor - 1]), & \\ \quad reduce\ f\ id\ (a[\lfloor \frac{|a|}{2} \rfloor \ldots |a| - 1]) & \text{otherwise} \end{cases}$$

What happens when f is the method for combining solutions?

```
reduce(merge, [], list(map(singleton, [1,3,6,4,8,7,5,2])))
```

This is Merge Sort! Can all divide-and-conquer algorithms be implemented with `reduce`?

The divide-and-conquer framework is much more general than `reduce`. So `reduce` cannot be used when, for example, we wish to split the input into 3 or more parts, or if they are of unequal size.

Maximum Continguous Subsequence Sum

Given a sequence of integers, the ***Maximum Contiguous Subsequence Sum Problem*** (MCSS) requires finding the contiguous subsequence of the sequence with maximum total sum:

$$\text{MCS}\,(a) = \arg \max_{0 \le i,j < |a|} \left(\left(\sum_{k=i}^{j} a[k] \right) \right).$$

We define the sum of an empty sequence to $-\infty$.

Example: For $a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \rangle$ a maximum contiguous subsequence (MCSS) is $\langle 3, -1, 0, 2 \rangle$. Another is $\langle 0, 3, -1, 0, 2 \rangle$.</p>

This is similar to Problem 4 on HW1 (`longest_run`). How?

MCSS is useful for many applications, from genomics to data science and finance.

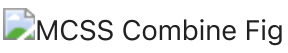
Let's first take a brute-force approach to this problem. What is the solution space, and how long does it take to evaluate it?

We must consider every contiguous subsequence and evaluate the sum of each. There are $O(n^2)$ contiguous subsequences. To evaluate the sum of each contiguous subsequence we need $O(n)$ work and $O(\log n)$ span. Thus the brute-force approach takes $O(n^3)$ work and $O(\log n)$ span.

Can we do better using divide-and-conquer?

As usual let's start by dividing the input into two equal parts and recursively finding the solution. If the MCSS is within either part entirely, then in the combine step we just need to return the subsequence with larger maximum.

But what if the MCSS spans the two halves?



Example: $a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \rangle$

- Split into left = $\langle 1, -2, 0, 3 \rangle$, right = $\langle -1, 0, 2, -3 \rangle$
- Left MCSS is $\langle 0, 3 \rangle$ (or just $\langle 3 \rangle$) $= 3$
- Right MCSS is $\langle 0, 2 \rangle = 2$
- But, the best MCSS crossing the cut is $\langle 0, 3, -1, 0, 2 \rangle = 4$

The maximum sum spanning the cut is the sum of the largest suffix on the left plus the largest prefix on the right.

Suppose we could identify an MCSS ending at position $\lfloor n/2 \rfloor$ and an MCSS beginning at position $\lfloor n/2 \rfloor$. Then we could add values of these to obtain a candidate MCSS for the whole sequence. Then the best of the three candidate solutions is an MCSS for the entire sequence.

- **MCSSS**(i): maximum contiguous sum **starting** at position i
- **MCSSE**(i): maximum contiguous sum **ending** at position i

Suppose MCSSE and MCSSS can be solved in $\Theta(n)$ work and $\Theta(\log n)$ span, and $bestAcross\ (b, c)$ constructs an MCSS crossing the split using these solutions. Then we could give this divide-and-conquer algorithm.

$$
\begin{array}{l}
MCSSDC\ a = \\
\quad \texttt{if}\ |a| = 0\ \texttt{then} \\
\quad\quad -\infty \\
\quad \texttt{else if}\ |a| = 1\ \texttt{then} \\
\quad\quad a[0] \\
\quad \texttt{else} \\
\quad\quad \texttt{let} \\
\quad\quad\quad (b, c) = splitMid\ a \\
\quad\quad\quad (m_b, m_c) = (MCSSDC\ b\ ||\ MCSSDC\ c) \\
\quad\quad\quad m_{bc} = bestAcross\ (b, c) \\
\quad\quad \texttt{in} \\
\quad\quad\quad \max\{m_b, m_c, m_{bc}\} \\
\quad\quad \texttt{end}
\end{array}
$$

Correctness:

We can proceed by induction as usual. The base case produces the correct results. For the induction step, we make the hypothesis that the recursively computed MCSS's for b and c are correct. With a correct implementation of $bestAcross$, we can conclude $\max\{m_b, m_c, m_{bc}\}$ is an MCSS.

Work/Span:

Using our assumption about $bestAcross$ we have that:

$$W(n) = 2W(n/2) + \Theta(n)$$

and

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

These yield $O(n \log n)$ work and $O(\log^2 n)$ span. What is a lower bound for the work? (It turns out we can match this lower bound.)

But how do we obtain an MCSS starting at a specified position (MCSSS) or ending at a specified position (MCSSE)?

Because one end of the contiguous subsequence is fixed, it turns out that we can use `reduce`, `scan` and `scanI` (an inclusive version of scan) to design algorithms for MCSSS and MCSSE using $O(n)$ work and $O(\log n)$ span for our application in which the start and end position is the middle of the input list.

MCSS starting at position i:

$$
\begin{array}{l}
MCSSSOpt\ a\ i = \\
\quad \textbf{let} \\
\quad\quad b = scanI\text{'+'}\ 0\ a\ [i \cdots (|a| - 1)] \\
\quad \textbf{in} \\
\quad\quad reduce\ max\ -\infty\ b \\
\quad \textbf{end}
\end{array}
$$

</p>

The intuition for this approach is just that if we lock position i for the subsequence, then computing a prefix sum and then subsequent maximum suffices.

 MCSSS Fig

left = $\langle 1, -2, 0, 3 \rangle$, right = $\langle -1, 0, 2, -3 \rangle$

```
>>> scan(add, 0, [-1,0,2,-3])
([-1, -1, 1, -2], -2)
```

Max of 1 happens when using prefix $[-1, 0, 2]$

Solving MCSSE is similar, except "backwards":

MCSS ending at position i:

$$
\begin{array}{l}
MCSSEOpt\ a\ j = \\
\quad \textbf{let} \\
\quad\quad (b, v) = scan\text{'+'}\ 0\ a[0 \cdots j] \\
\quad\quad w = reduce\ min\ \infty\ b \\
\quad \textbf{in} \\
\quad\quad v - w \\
\quad \textbf{end}
\end{array}
$$

Here, v is the sum of $a[0 \cdots j]$. The key observation here is that for a location i, if we subtract prefix sum up to position i from v then we have the subsequence sum for $a[i \cdots j]$. The first `scan` computes these prefixes and the `reduce` identifies which difference is maximum.

 MCSSE Fig

```
>>> b, v = scan(add, 0, [1,-2,0,3])
>>> print('b=', b, 'v=', v)
b= [1, -1, -1, 2] v= 2

>>> w = reduce(min, -math.inf, b)
>>> print('w=', w)
w= -1

>>> print('v-w=', v-w)
v-w= 3
```

Max of 3 happens for subsequence $[0, 3]$

In [17]:

```
import math

def reduce(f, id_, a):
    # print('a=%s' % a) # for tracing
```

```
    if len(a) == 0:
        return id_
    elif len(a) == 1:
        return a[0]
    else:
        # can call these in parallel
        return f(reduce(f, id_, a[:len(a)//2]),
                 reduce(f, id_, a[len(a)//2:]))

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

def scan(f, id_, a):
    """
    This is a horribly inefficient implementation of scan
    only to understand what it does.
    We'll discuss how to make it more efficient later.
    """
    return (
            [reduce(f, id_, a[:i+1]) for i in range(len(a))],
             reduce(f, id_, a)
           )

def MCSS_prefix(a):
    # return the MCSS in a that starts at index i
    b = scan(add, 0, a)
    return reduce(max, -math.inf, b[0])

def MCSS_suffix(a):
    # return the MCSS in a that ends at index j
    b = list(a)
    b.reverse()
    return MCSS_prefix(b)

def best_across(b, c):
    # return the MCSS of a sequence that crosses input sequences b and c
    return MCSS_suffix(b) + MCSS_prefix(c)

def MCSS(a):
    if len(a) == 0:
        return -math.inf
    elif len(a) == 1:
        return a[0]
    else:
        b = a[:len(a)//2]
        c = a[len(a)//2:]
        sum_b = MCSS(b)
        sum_c = MCSS(c)
        sum_across = best_across(b,c)
        return max(sum_b, sum_c, sum_across)

left = [1,-2,0,3]
right = [-1,0,2,-3]
MCSS(left + right)
```

Out[17]:
```
4
```

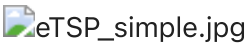
The Euclidean Traveling Salesperson Problem

In the Euclidean Traveling Salesperson Problem (eTSP), you are given a set of n 2D points. The goal is to find a "tour" (of the points with minimum cost. That is, we must construct a sequence of all the points (i.e., a sequence of 2D points) that begins and ends with the same point such that:

- every point is visited exactly once (except the starting point)
- the sum of distances between adjacent points is minimized

This is an incredibly widespread and useful problem -- consider all the various kinds of routing problems that are solved every day. For a simple example, think of Amazon/USPS/UPS package deliveries.

Which solution is better?



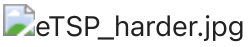
Given an input with n points, how many possible solutions are there?

Brute-Force?

If we take a brute-force approach to this problem, what is the solution space and how can we search it?

There are $n!$ possible solutions, and we must check the cost of each by summing $n-1$ distances. This can be done with $O(n)$ work and $O(\log n)$ span. So we can solve eTSP with $O(n \cdot n!)$ work and $O(\log n)$ span.

This is good span, but an astronomical amount of work. What if we had more points?



16! is about $2x10^{13}$, so while there are very few points the brute-force approach is not tractable!

Is the brute-force algorithm work-efficient?

Divide-and-Conquer?

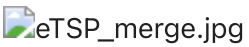
What intuition can we get about the fact that this problem is in 2D?

Since points that are "clustered" can possibly be dealt with first, how about a divide-and-conquer approach? How would that work?

We can split the input using a "cut" through the plane that separates the input points into two equal parts. Then, recursively solve eTSP for each smaller point set.

How do we combine smaller solutions into larger ones?

We need to make sure that two tours can be combined into the best possible single tour.



To do this, we can try all possible ways to merge each tour by rerouting across the cut and back and choose the least costly. This yields the following algorithm:

$$
\begin{array}{l}
eTSP\,(P) = \\
\quad \texttt{if } |P| < 2 \texttt{ then} \\
\quad\quad \texttt{raise } TooSmall \\
\quad \texttt{else if } |P| = 2 \texttt{ then} \\
\quad\quad \langle\, (P[0], P[1]), (P[1], P[0]) \,\rangle \\
\quad \texttt{else} \\
\quad\quad \texttt{let} \\
\quad\quad\quad (P_\ell, P_r) = split\ P \texttt{ along the longest dimension} \\
\quad\quad\quad (L, R) = (eTSP\ P_\ell) \ ||\ (eTSP\ P_r) \\
\quad\quad\quad (c, (e, e')) = minVal_{first}\,\{(swapCost(e, e'), (e, e')) : e \in L, e' \in R\} \\
\quad\quad \texttt{in} \\
\quad\quad\quad\quad swapEdges\ (append\ (L, R), e, e') \\
\quad\quad \texttt{end}
\end{array}
$$

The function $minVal_{first}$ uses the first value of the pairs to find the minimum, and returns the (first) pair with that minimum. The function $swapEdges(E, e, e')$ finds the edges e and e' and swaps the endpoints. As there are two ways to swap, it picks the cheaper one.

Correctness: Does this algorithm compute a tour? Does this algorithm compute a minimum-cost tour?

We can show by induction that this algorithm always produces a tour.

However, the combine step does not necessarily produce a minimum cost tour!

Actually, we currently do not know of any polynomial-work algorithm to solve this problem. In fact, the brute-force algorithm is essentially the best we can do. (We'll get to this in more detail at the end of the semester.)

What we do know how to do efficiently is to compute an *approximation* to the optimal eTSP solution. We can compute a solution that is within $(1 + \epsilon)$ of optimal. The running time is polynomial in n and $1/\epsilon$.

This algorithm is actually not correct in the sense that it is not necessarily an approximation to the optimal solution. Rather, it is a *heuristic* that works well in practice.

Work/Span:

This algorithm has two recursive calls that each operate on $n/2$ points. To combine the solution we must check $O(n^2)$ ways too cross the cut and compute the best. This requires $O(n^2)$ work and $O(\log n)$ span.

So we have that the work is $W(n) = 2W(n/2) + O(n^2)$. This is a root-dominated recurrence, and thus $W(n) = O(n^2)$.

The span is $S(n) = S(n/2) + O(\log n)$. This is a balanced recurrence with $\lg n$ levels, and so $S(n) = O(\log^2 n)$.