We have already seen a couple of divide and conquer algorithms in this lecture. The reduce algorithm and the algorithm to copy elements of the array are both D&C algorithms. We have also seen a divide and conquer matrix multiplication algorithm. We will use this technique a lot in this class, so let's formalize it a bit.

Divide and conquer algorithms generally have 3 steps: divide the problem into subproblems, recursively solve the subproblems and combine the solutions of subproblems to create the solution to the original problem.

The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction. This makes it easy to show correctness and also to figure out cost bounds. The general structure looks as follows:

— **Base Case:** When the problem is sufficiently small, return the trivial answer directly or resort to a different, usually simpler, algorithm, which works great on small instances.

— **Inductive Step:** First, the algorithm divides the current instance $I$ into parts, commonly referred to as subproblems, each smaller than the original problem. Then, it recurses on each of the parts to obtain answers for the parts. In proofs, this is where we assume inductively that the answers for these parts are correct, and based on this assumption, it combines the answers to produce an answer for the original instance $I$.

This technique is even more useful for parallel algorithms. Generally, you can solve the subproblems in parallel. If the divide and combine step is inexpensive, then you are done. If either the divide and combine step (or both) is expensive, however, you may want to parallelize it (them), which can be difficult, depending on the algorithms.

Let us assume that the subproblems can be solved independently. Say the problem of size $n$ is broken into $k$ subproblems of size $n_1, \ldots, n_k$. How would you write the Cilk Plus program? What if $k = 2$?
F(n)
1  if n ≤ n₀
2    then Base-Case
3    return
4  Divide into 2 parts of size n₁ and n₂
5  spawn F(n₁)
6  F(n₂)
7  sync
8  Combine.

With the strategy above, the work is
\[
W(n) = W_{\text{divide}}(n) + W(n₁) + W(n₂) + W_{\text{combine}}(n)
\]

And the span is
\[
S(n) = S_{\text{divide}}(n) + \max\{S(n₁), S(n₂)\} + S_{\text{combine}}(n)
\]

Note that the work recurrence is simply adding up the work across all components. More interesting is the span recurrence: First, note that a divide and conquer algorithm has to split a problem instance into subproblems before these subproblems are recursively solved. Furthermore, it cannot combine the results from these subproblems to generate the ultimate answer until the recursive calls on the subproblems are complete. This forms a chain of sequential dependencies, explaining why we add their span together. The parallel execution takes place among the recursive calls since we assume that the subproblems can be solved independently — this is why we take the max over the subproblems’ span.

Now consider arbitrary k. What is the pseudocode?

F(n)
1  if n ≤ n₀
2    then Base-Case
3    return
4  Divide into k parts of size n₁, n₂, ..., nₖ
5  parallel_for i ← 1 to k
6    do F(nᵢ)
7  Combine.

\[
W(n) = W_{\text{divide}}(n) + \sum_{i=1}^{k} W(nᵢ) + W_{\text{combine}}(n)
\]
And the span is
\[ S(n) = S_{\text{divide}}(n) + \frac{1}{2} \log k + \max_{i=1}^{k} \{ S(n_i) + S_{\text{combine}}(n) \} \]

Applying this formula results in familiar recurrences such as \( W(n) = 2W(n/2) + O(n) \). In
the rest of this lecture, we’ll get to see other recurrences—and learn how to derive a closed-form
for them.

1 Maximum contiguous subsequence sum problem (MCSS)

In most divide-and-conquer algorithms you have encountered so far, the subproblems are occur-
rences of the problem you are solving. This is not always the case. Often, you’ll need more
information from the subproblems to properly combine results of the subproblems. In this case,
you’ll need to strengthen the problem, much in the same way that you strengthen an inductive hy-
pothesis when doing an inductive proof. Let’s take a look at an example: the maximum contiguous
subsequence sum (MCSS) problem. MCSS can be defined as follows:

Definition 1 (The Maximum Contiguous Subsequence Sum (MCSS) Problem) Given a sequence
of numbers \( s = \langle s_1, \ldots, s_n \rangle \), the maximum contiguous subsequence sum problem is to find
\[ \max \left\{ \sum_{k=i}^{j} s_k : 1 \leq i \leq n, i \leq j \leq n \right\} \].
(i.e., the sum of the contiguous subsequence of \( s \) that has the largest value).

For example, the MCSS of a sequence \( \langle 2, -5, 4, 1, -2, 3 \rangle \) is 6, via the subsequence \( \langle 4, 1, -2, 3 \rangle \).

Algorithm 1: Brute Force

The brute force algorithm examines all possible combinations of subsequences and for each one
of them, it computes the sum and takes the maximum. Note that every subsequence of \( s \) can be
represented by a starting position \( i \) and an ending position \( j \). We will use the shorthand \( s_{i..j} \) to
denote the subsequence \( \langle s_i, s_{i+1}, \ldots, s_j \rangle \).

MCSS[1..n]
1 parallel for all tuples \( (i, j) \) such that \( i \leftarrow 1 \) to \( n \) and \( j \leftarrow i \) to \( n \)
2 do \( A[i, j] \leftarrow \text{spawn} \text{SUM}(i, j) \)
3 \text{MAX}(A)
The total work is $O(n^3)$. We have learned in last lecture that we can use reduction to compute $\text{SUM}(i, j)$ in parallel, with the work of $\theta(n)$ and span of $\theta(\lg n)$. Similarly, you can compute $\text{MAX}$ using reduction with the same asymptotic work and span. That means, the total work of this Brute Force algorithm is $\theta(n^3)$, and the span is $\theta(\lg n)$.

**Exercise 1** Can you improve the work of the naïve algorithm to $O(n^2)$? What does this do the span?

**Algorithm 2: Divide And Conquer — Version 1.0**

We’ll design a divide-and-conquer algorithm for this problem.

- **Divide:** Split the sequence in half.
- **Conquer:** Recursively solve for both halves.
- **Combine:** This is the most interesting step. For example, imagine we split the sequence in the middle and we get the following answers:

\[
\langle \ldots \mid L \mid R \ldots \rangle \\
\begin{align*}
L &= \langle \ldots \rangle_{\text{mcss}=56} \\
R &= \langle \ldots \rangle_{\text{mcss}=17}
\end{align*}
\]

There are 3 possibilities: (1) the maximum sum lies completely in the left subproblem, (2) the maximum sum lies completely in the right subproblem, and (3) the maximum sum spans across the split point. The first two cases are easy. The more interesting case is when the largest sum goes between the two subproblems. The maximum subsequence that spans the middle is equal to the largest sum of a suffix on the left and the largest sum of a prefix on the right.

**MCSS**($S, i, j$)

1. **if** $i = j$
2. **then** return $S[i]$
3. mid $\leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
4. $L \leftarrow \text{spawn} \text{MCSS}(S, i, \text{mid})$
5. $R \leftarrow \text{spawn} \text{MCSS}(S, \text{mid} + 1, j)$
6. sync
7. $LS \leftarrow \text{spawn} \text{MAXSUFFIXSUM}(S, i, \text{mid})$
8. $RP \leftarrow \text{MAXPREFIXSUM}(S, \text{mid} + 1, j)$
9. sync
10. **return** max($L, LS + RP, R$)
How would you calculate the suffix and the prefix? If you do it in the naive sequential way (e.g. keep the running prefix sum, and update the best as you encounter each element), then its work and span is \( \theta(n) \). Then the work of the MCSS algorithm is

\[
T_1(n) = 2T_1(n/2) + \theta(n) = \theta(n \log n)
\]

and the span is

\[
T_\infty(n) = T_\infty(n/2) + \theta(n) = \theta(n)
\]

This is not great, since the parallelism is only \( \theta(\log n) \). Note that you don’t have to sync between calculating the recursive steps and calculating the prefix and suffix. However, it will not help with the span if you didn’t. We will show next week how to compute the max prefix and suffix sums in parallel, but for now, we’ll take it for granted that they can be done in \( \theta(n) \) work and \( \theta(\log n) \) span. This reduces the span to

\[
T_\infty(n) = T_\infty(n/2) + \theta(\log n) = \theta(\log^2 n)
\]

**Exercise 2** Solve the work and span recurrences without using the master method. That is, use the recursion tree method you learned in CSE 241. Also prove the bounds using induction, as you learned in CSE 241.

**Algorithm 3: Divide And Conquer — Version 2.0**

As it turns out, we can do better than \( O(n \log n) \) work. The key is to strengthen the (sub)problem—i.e., solving a problem that is slightly more general—to get a faster algorithm. Looking back at our previous divide-and-conquer algorithm, the “bottleneck” is that the combine step takes linear work. Is there any useful information from the subproblems we could have used to make the combine step take constant work instead of linear work?

In the design of our previous algorithm, we took advantage of the fact that if we know the max suffix sum and max prefix sums of the subproblems, we can produce the max subsequence sum in constant time. The expensive part was in fact computing these prefix and suffix sums—we had to spend linear work because we didn’t know how generate the prefix and suffix sums for the next level up without recomputing these sums. *Can we easily fix this?*

The idea is to return the overall sum together with the max prefix and suffix sums, so we return a total of 4 values: the max subsequence sum, the max prefix sum, the max suffix sum, and the overall sum. Having this information from the subproblems is enough to produce a similar answer tuple for all levels up, in constant work and span per level. More specifically, *we strengthen our problem to return a 4-tuple \( \text{mcss, max-prefix, max-suffix, total} \), and if the recursive calls return \( (m_1, p_1, s_1, t_1) \) and \( (m_2, p_2, s_2, t_2) \), then we return \( (\max(s_1 + p_2, m_1, m_2), \max(p_1, t_1 + p_2), \max(s_1 + t_2, s_2), t_1 + t_2) \)*.
Here’s the code for this algorithm:

\[
\text{MCSS}(A, i, j)
\]

1. if \(i = j\)
2. \(m \leftarrow \max\{A[i], 0\}, p \leftarrow \max\{A[i], 0\}, s \leftarrow \max\{A[i], 0\}, t \leftarrow A[i]\)
3. \(\text{return } (m, p, s, t)\)
4. \(\text{mid } \leftarrow \left\lfloor \frac{i + j}{2} \right\rfloor\)
5. \((m_1, p_1, s_1, t_1) \leftarrow \text{spawn MCSS}(A, i, \text{mid})\)
6. \((m_2, p_2, s_2, t_2) \leftarrow \text{MCSS}(A, \text{mid} + 1, j)\)
7. \(\text{sync}\)
8. \(m \leftarrow \max\{s_1 + p_2, m_1, m_2\}, p \leftarrow \{p_1, t_1 + p_2\}, s \leftarrow \{s_1 + t_2, s_2\}, t \leftarrow t_1 + t_2\)
9. \(\text{return } (m, p, s, t)\)

**Cost Analysis.** We have the recurrences

\[
W(n) = 2W(n/2) + O(1) = O(n)
\]

\[
S(n) = S(n/2) + O(1) = O(\log n)
\]

## 2 Parallel Mergesort

We now talk about merge sort. Recall that in the first lecture, we saw a parallel version of merge sort — here we do the obvious thing and make the recursive calls in parallel.

\[
\text{MergeSort}(A, n)
\]

1. if \(n = 1\)
2. \(\text{then return } A\)
3. \(\text{Divide } A \text{ into two } A_{\text{left}} \text{ and } A_{\text{right}} \text{ each of size } n/2\)
4. \(A'_{\text{left}} \leftarrow \text{spawn MergeSort}(A_{\text{left}}, n/2)\)
5. \(A'_{\text{right}} \leftarrow \text{MergeSort}(A_{\text{right}}, n/2)\)
6. \(\text{sync}\)
7. \(\text{Merge the two halves into } A'\)
8. \(\text{return } A'\)

The work of the algorithm remains unchanged. What is the span? The recurrence is

\[
S_{\text{MergeSort}}(n) = S_{\text{MergeSort}}(n/2) + S_{\text{merge}}(n)
\]

Since we are merging the arrays sequentially, the span of the merge call is \(\Theta(n)\) and the recurrence solves to \(S_{\text{MergeSort}}(n) = \Theta(n)\). Therefore, the parallelism of this merge sort operation is \(\Theta(\log n)\),
which is very small. In general, you want the parallelism to be polynomial in \( n \), not logarithmic in \( n \).

What is the problem? It is the merge operation — doing merge sequentially is the bottleneck.

3 Let’s Parallelize the Merge in Mergesort

In Mergesort, we generally merge two arrays of the same size. However, in order to get this parallel merge to work, we have to be more general. We must learn how to merge two arrays which can be different in size.

**Problem Statement**: Given two arrays \( B[1..m] \) and \( C[1..l] \), each of which is sorted, we want to merge them into a sorted array \( A[1..n] \) where \( n = m + l \). Without loss of generality say that \( m > l \). Here’s the procedure.

```
ParallelMerge(B, m, C, l)
1   if m < l
2      then return MERGE(C, l, B, m)
3   if m = 1,
4      then Concatenate the arrays in the right order and return.
5     mid ← ⌊m/2⌋
6     s ← SEARCH(C, B[mid]).
7     A'_{left} ← spawn MERGE(B[1..mid], mid, C[1..s], s)
8     A'_{right} ← spawn MERGE(B[mid + 1..m], m - mid, C[s + 1..l], l - s)
9     sync
10    Concatenate A'_{left} and A'_{right} and return
```

Let us calculate work and span. The search takes \( \Theta(lg n) \) work and span.

Say \( k = mid + s \). First, we notice that

\[
\begin{align*}
k &= mid + s \\
 &= m/2 + s \\
& \geq m/2 + 0 \\
& \geq n/4 \\
& = n/4
\end{align*}
\]

Since \( k \) and \( n - k \) are symmetric, we have \( n/4 \leq k \leq 3n/4 \).

\[
\begin{align*}
W_{Merge}(n) &= W_{Merge}(k) + W_{Merge}(n - k) + \Theta(lg n) \\
&= W_{Merge}(\alpha n) + W_{Merge}((1 - \alpha)n) + \Theta(lg n) \text{ for some } 1/4 \leq \alpha \leq 3/4 \\
&= \Theta(n)
\end{align*}
\]
Exercise 3 Draw the recursion tree for the above recurrence to solve it.

Exercise 4 Show using induction that the recurrence \( W(n) = W(\alpha n) + W((1 - \alpha)n) + \Theta(\lg n) \) solves to \( \Theta(n) \).

For span, we have:

\[
S_{\text{Merge}}(n) = \max\{S_{\text{Merge}}(k), S_{\text{Merge}}(n - k)\} + \Theta(\lg n)
\leq S_{\text{Merge}}(3n/4) + \Theta(\lg n)
= \Theta(\lg^2 n)
\]

Note that parallelizing the Merge procedure did not increase its work — which is exactly what we want. It is a work-efficient algorithm. But we reduced the span from \( \Theta(n) \) to \( \Theta(\lg^2 n) \).

Work and Span of Parallel Merge Sort using Parallel Merge

We can use this parallel merge procedure as a subroutine of merge sort and our work remains \( \Theta(n \lg n) \). If we substitute the span of merge back into the Merge Sort equation, we get

\[
S_{\text{MergeSort}}(n) = S_{\text{MergeSort}}(n/2) + S_{\text{Merge}}(n)
= S_{\text{MergeSort}}(n/2) + \Theta(\lg^2 n)
= \Theta(\lg^3 n)
\]

Therefore, the parallelism of this new merge sort procedure is \( \Theta(n \lg n / \lg^3 n) = \Theta(n / \lg^2 n) \). Therefore, we now have polynomial amount of parallelism.