Lecture 3 — Algorithmic Techniques and Divide-and-Conquer

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 4, 2012

Material in this lecture:

- Quick review of Cost Models (from last lecture)
- Algorithmic techniques
- Divide and Conquer

1 Algorithmic Techniques

There are many algorithmic techniques/approaches for solving problems. Understanding when and how to use these techniques is one of the most important skills of a good algorithms designer. In the context of the shortest superstring (SS) problem we already mentioned three techniques: brute force, reducing one problem to another, and the greedy approach. In this course we will cover many techniques and we now give a brief overview of the techniques. We will then go into one of the techniques, divide-and-conquer, in more detail. All these techniques are useful for both sequential and parallel algorithms, however some, such as divide-and-conquer, play an even larger role in parallel algorithms than sequential algorithms.

Brute Force: The brute force approach typically involves trying all possibilities. In the SS problem, for example, we argued that every solution has to correspond to a permutation of the inputs with overlaps removed. The brute force approach therefore tried all permutations and picked the best. Since there are $n!$ permutations, this solution is not “tractable” for large problems. In many other problems there are only polynomially many possibilities. For example in your current assignment there should be only $O(n^2)$ possibilities to try. However, even $O(n^2)$ is not good, since as you will work out in the assignment there is another solution that require only $O(n)$ work. One place the brute force approach can be very useful is when writing a test routine to check the correctness of more efficient algorithms. Even if inefficient for large n the brute force approach could work well for testing small inputs. The brute force approach is often the simplest solution to a problem, although not always.

Reducing to another problem: Sometimes the easiest approach to solving a problem is just reduce it to another problem for which known algorithms exist. For the SS problem we reduced it to what would seem superficially to be a very different problem, the Traveling Salesperson (TSP) problem. The reduction was pretty straightforward, but how would anyone even think of doing such a translation. When you get more experienced with algorithms you will start recognizing similarities between problems that on the surface seem very different.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

Inductive techniques: The idea behind inductive techniques is to solve one or more smaller problems that can be used to solve the original problem. The technique most often uses recursion to solve the sub problems and can be proved correct using (strong) induction. Common techniques that fall in this category include:

- *Divide-and-conquer.* Divide the problem on size n into $k > 1$ subproblems on sizes $n_1, n_2, \ldots n_k$, solve the problem recursively on each, and combine the solutions to get the solution to the original problem.
- *Greedy.* For a problem on size n use some greedy approach to pull out one element leaving a problem of size $n - 1$. Solve the smaller problem.
- *Contraction.* For a problem of size n generate a significantly smaller (contracted) instance (e.g. of size $n/2$), solve the smaller instance, and then use the result to solve the original problem. This only differs from divide and conquer in that we make one recursive call instead of multiple.
- *Dynamic Programming.* Like divide and conquer, dynamic programming divides the problem into smaller problems and then combines the solutions to the subproblems to find a solution to the original problem. The difference, though, is that the solutions to subproblems are reused multiple times. It is therefore important to store the solutions for reuse either using memoization or by building up a table of solutions.

Data Types and Data Structures: Some techniques make heavy use of the operations on abstract data types representing collections of values and their implementation using a variety of data structures. Abstract collection types that we will cover in this course include: Sequences, Sets, Priority Queues, Graphs, and Sparse Matrices.

Randomization: Randomization in is a powerful technique for getting simple solutions to problems. We will cover a couple of examples in this course. Formal cost analysis for many randomized algorithms, however, requires probability theory beyond the level of this course.

Once you have defined the problem, you can look into your bag of techniques and, with practice, you will find a good solutions to the problem. When we say a *good solution* we mean:

1. **Correct:** Clearly correctness is most important.
2. **Low cost:** Out of the correct solutions, we would prefer the one with the lowest cost.

2 Divide and Conquer

Divide and conquer is a highly versatile technique that generally lends itself very well to parallel algorithms design. You probably have seen the divide-and-conquer technique many times before this course, but it is such an important technique that it is worth seeing it over and over again. It is particularly suited for “thinking parallel” because it offers a natural way of creating parallel tasks.

In most divide-and-conquer algorithms you have encountered so far, the subproblems are occurrences of the problem you are solving (e.g. recursive sorting in merge sort). 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 definition, much in the same way that you strengthen an inductive hypothesis when doing an inductive proof. This involves defining a problem that solves more than what you ultimately need, but makes it easier or even possible to use solutions of subproblems to solve the larger problem. In the recitation you looked at how to solve the Parenthesis Matching problem by defining a version of the problem that returns the number of unmatched parentheses on the right and left. This is a stronger problem than the original—the original is the special case when both these values are zero (no unmatched right or left parentheses). Therefore the modified version tells us more than we need but was necessary to make the divide-and-conquer work.

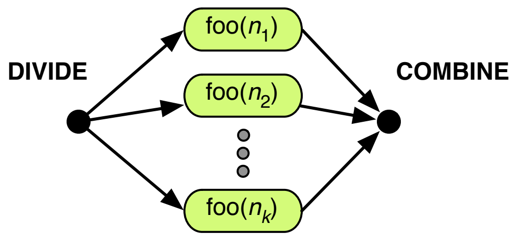
The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction. This makes it easy to show correctness. Often it also makes it easy to determine costs bounds since we can write recurrences that match the inductive structure. The general form of divide-and-conquer looks as follows:

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

— **Inductive Step:**

1. First, the algorithm *divides* the current instance I into parts, commonly referred to as *subproblems*, each smaller than the original problem.
2. It then 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.
3. It then *combines* the answers to produce an answer for the original instance I, and in the reasoning about correctness and proof we have to show that this combination gives the correct answer.

This process can be schematically depicted as



On the assumption that the subproblems can be solved independently, the work and span of such an algorithm can be described using simple recurrences. In particular for a problem of size n is broken into k subproblems of size $n_1, \ldots, n_k$, then the work is

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

and the span is

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

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. We therefore have to add the span for the divide step. The algorithm can then execute all the subproblems in parallel. We therefore take the maximum of the span for these subproblems. Finally *after* all the problems complete we can combine the results. We therefore have to add in the span for the combine step.

Applying this formula often 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. For how, let's derive a closed-form for this expression.

The first recurrence we're looking at is $W(n) = 2W(n/2) + O(n)$, which you probably have seen many times already. To derive a closed form for it, we'll review the tree method, which you have seen in 15-122 and 15-251.

But first, let's think through what it means when we write $O(f(n))$ in an expression (e.g., when we write $4W(n/2) + O(n)$ in the recurrence above). In these expression, we write $O(f(n))$ in place of *some* function $g(n) \in O(f(n))$. From the definition of $O(\cdot)$, this means that there exist positive constants N_0 and c such that for all $n \geq N_0$, we have $g(n) \leq c \cdot f(n)$. It follows that *there exist constants k_1 and k_2 such that for all $n \geq 1$,*

$$g(n) \;\leq\; k_1 \cdot f(n) + k_2,$$

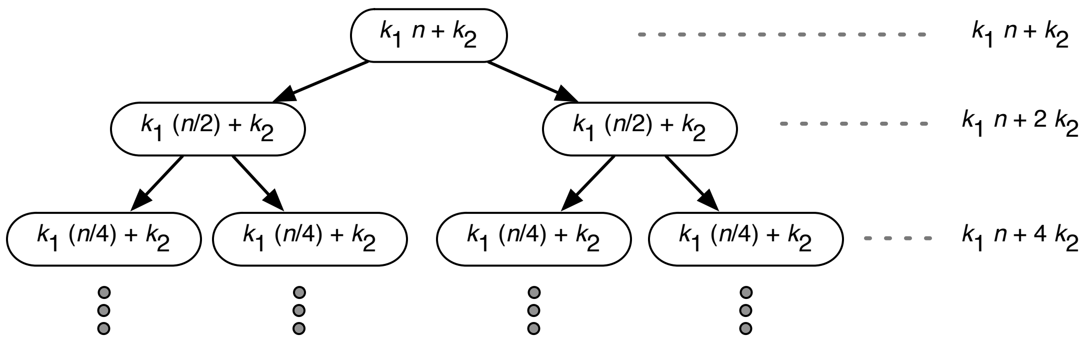
where, for example, we can take $k_1 = c$ and $k_2 = \sum_{i=1}^{N_0} |g(i)|$.

By this argument, we can establish that

$$W(n) \quad\leq\quad 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants. We'll now use the tree method.

The idea of the tree method is to consider the recursion tree of the algorithm in hope to derive an expression for what's happening at each level of the tree. In this particular example, we can see that each node in the tree has 2 children, whose input is half the size of that of the parent node. Moreover, if the input has size n, the recurrence shows that the cost, excluding that of the recursive calls, is at most $k_1 \cdot n + k_2$. Therefore, our recursion tree annotated with cost looks like this:



To apply the tree method, there are some key questions we should ask ourselves to aid drawing out the recursion tree and to understand the cost associated with the nodes:

- How many levels are there in the tree?
- What is the problem size at level i?
- What is the cost of each node in level i?
- How many nodes are there at level i?
- What is the total cost across level i?

Our answers to these questions lead to the following analysis: We know that level i (the root is level $i = 0$) contains 2^i nodes, each costing at most $k_1(n/2^i) + k_2$. Thus, the total cost in level i is at most

$$2^i \cdot \left(k_1 \frac{n}{2^i} + k_2\right) \quad = \quad k_1 \cdot n + 2^i \cdot k_2.$$

Since we keep halving the input size, the number of levels is bounded by $1 + \log n$. Hence, we have

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log n} \left(k_1 \cdot n + 2^i \cdot k_2\right) \\ &= k_1 n(1 + \log n) + k_2(n + \tfrac{n}{2} + \tfrac{n}{4} + \cdots + 1) \\ &\leq k_1 n(1 + \log n) + 2k_2 n \\ &\in O(n \log n), \end{aligned}$$

where in the second to last step, we apply the fact that for $a > 1$,

$$1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \leq a^{n+1}.$$

2.1 A Useful Trick — The Brick Method

The tree method involves determining the depth of the tree, computing the cost at each level, and summing the cost across the levels. Usually we can easily figure out the depth of the tree and the cost of at each level relatively easily—but then, the hard part is taming the sum to get to the final answer. The following trick can help you derive the final answer quickly. By recognizing which shape a given recurrence is, we can almost immediately determine the asymptotic complexity of that recurrence.

The vital piece of information is *the ratio of the cost between adjacent levels*. Let cost_i denotes the total cost at level i when we draw the recursion tree. This puts recurrences into three broad categories:

Leaves Dominated	***Balanced***	***Root Dominated***
Each level is larger than the level before it by at least a constant factor. That is, there is a constant $\rho > 1$ such that for all level i, $\mathsf{cost}_{i+1} \geq \rho \cdot \mathsf{cost}_i$	All levels have approximately the same cost.	Each level is smaller than the level before it by at least a constant factor. That is, there is a constant $\rho < 1$ such that for all level i, $\mathsf{cost}_{i+1} \leq \rho \cdot \mathsf{cost}_i$
++ ++++ ++++++ ++++++++	++++++++ ++++++++ ++++++++ ++++++++	+++++++++ ++++++ ++++ ++
Implication: $O(\mathsf{cost}_d)$, where d is the depth	*Implication:* $O(d \cdot \max_i \mathsf{cost}_i)$	*Implication:* $O(\mathsf{cost}_0)$
The house is stable, with a strong foundation.	The house is sort of stable, but don't build too high.	The house will tip over.

You might have seen the "master method" for solving recurrences in previous classes. We do not like to use it since it only works for special cases and does not give an intuition of what is going on. However, we will note that the three cases of the master method correspond to special cases of leaves dominated, balanced, and root dominated.

3 An Example : The Maximum Contiguous Subsequence Sum Problem

We now consider an example problem for which we can find a couple divide-and-conquer solutions.

Definition 3.1 (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

$$\mathsf{mcss}(s) = \max \left\{ \sum_{k=i}^{j} s_k \;:\; 1 \leq i \leq n, i \leq j \leq n \right\}.$$

(i.e., the sum of the contiguous subsequence of s that has the largest value).

Let's look at an example. For $s = \langle 1, -5, 2, -1, 3 \rangle$, we know that $\langle 1 \rangle$, $\langle 2, -1, 3 \rangle$, and $\langle -5, 2 \rangle$ are all *contiguous* subsequences of s—whereas $\langle 1, 2, 3 \rangle$ is not. Among such subsequences, we're interested in finding one that maximizes the sum. In this particular example, we can check that $\mathsf{mcss}(s) = 4$, achieved by taking the subsequence $\langle 2, -1, 3 \rangle$.

We have to be careful about what our MCSS problem returns given an empty sequence as input. Here we assume the max of an empty set is $-\infty$ so it returns $-\infty$. We could alternatively decide it is undefined.

3.1 Algorithm 1: Brute Force

Immediate from the definition is an algorithm with $O(n^3)$ work and $O(\log n)$ span. This algorithm examines all possible contiguous subsequences and for each one of them, it computes the sum. It then takes the maximum of these sums. Note that every subsequence of s can be represented by a starting position i and an ending position j.

The pseudocode for this algorithm using our notation exactly matches the definition of the problem. For each subsequence $i..j$, we can compute its sum by applying a plus `reduce`. This does $O(j-i)$ work and has $O(\log(j-i))$ span. Furthermore, all the subsequences can be examined independently in parallel (using, e.g., `tabulate`). This leads to the following bounds:

$$\begin{aligned} W(n) &= 1 + \sum_{1\le i\le j\le n} W_{\text{reduce}}(j-i) \le 1 + n^2 \cdot W_{\text{reduce}}(n) = 1 + n^2 \cdot O(n) = O(n^3) \\ S(n) &= 1 + \max_{1\le i\le j\le n} S_{\text{reduce}}(j-i) \le 1 + S_{\text{reduce}}(n) = O(\log n) \end{aligned}$$

These are cost bounds for enumerating over all possible subsequences and computing their sums. The final step of the brute-force solution is to find the max over these combinations. Since max reduce has $O(n^2)$ work and $O(\log n)$ span[1], the cost of the final step is subsumed by other costs analyzed above. Overall, we have an $O(n^3)$-work $O(\log n)$-span algorithm.

As you might notice already, this algorithm is clearly inefficient. We'll apply divide and conquer to come up with a more efficient solution.

3.2 Algorithm 2: Divide And Conquer — Version 1.0

We'll design a divide-and-conquer algorithm for this problem. In coming up with such an algorithm, an important first step lies in figuring out how to properly divide up the input instance.

What is the simplest way to divide a sequence? Let us divide the sequence in half, recursively solve the problem on both halves and see what we can do. For example, imagine we divide the sequence in the middle and we get the following answers:

$$\langle \text{——}\ L\ \text{——} \parallel \text{——}\ R\ \text{——} \rangle$$
$$\Downarrow$$
$$L = \underbrace{\langle\ \cdots\ \rangle}_{\text{mcss}=56} \qquad R = \underbrace{\langle\ \ldots\ \rangle}_{\text{mcss}=17}$$

How would we use this to create a divide-and-conquer algorithm? So far, we have suggested a simple way to divide the input sequence. Assuming that we can recursively solve the problem, by calling the algorithm itself on smaller instances, we need answer the next question, how to combine the answers to the subproblems to obtain the final answer?

Our first (blind) guess might be to return $\max(\mathsf{mcss}(L), \mathsf{mcss}(R))$. This is unfortunately incorrect. We need a better understanding of the problem to devise a correct combine step. Notice that the

[1]Note that it takes the maximum over $\binom{n}{2} \le n^2$ values, but since $\log n^a = a \log n$, this is simply $O(\log n)$

subsequence we're looking for has one of the three forms: (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 divide point. The first two cases are easy and have already been taken care of by the recursive calls. The more interesting case is when the largest sum goes between the two subproblems.

How do we tackle this case? It is not hard to see that the maximum sum going across the divide is the largest sum of a suffix on the left and the largest sum of a prefix on the right. Cost aside, this leads to the following algorithm:

```
fun mcss(s) =
  case (showt s)
    of EMPTY = $-\infty$
     | ELT(x) = x
     | NODE(L,R) =
        let val ($m_L, m_R$) = (mcss(L) || mcss(R))
            val $m_A$ = bestAcross(L,R)
        in max{$m_L, m_R, m_A$}
        end
```

We will show you soon 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 $O(n)$ work and $O(\log n)$ span. We'll also assume that dividing takes $O(\log n)$ work and span (as you have seen in 15-150). This yields the following recurrences:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ S(n) &= S(n/2) + O(\log n) \end{aligned}$$

Proof of correctness. Let's switch gear and talk about proof of correctness. So far, as was the case in 15-150, you are familiar with writing detailed proofs that reason about essentially every step down to the most elementary operations. In this class, although we still expect your proof to be rigorous, we are more interested in seeing the critical steps highlighted and less important or standard ones summarized, with the idea being that if probed, you can easily provide detail on demand. The idea is that we want to make key ideas in an algorithm stand out as much as we can. It will be difficult for us to specify exactly how detailed we expect the proof to be, but you will pick it up by example.

We'll now prove that the `mcss` algorithm above computes the maximum contiguous subsequence sum. Specifically, we're proving the following theorem:

Theorem 3.2. *Let s be a sequence. The algorithm* mcss(s) *returns the maximum contiguous subsequence sum in s—and returns* $-\infty$ *if s is empty.*

Proof. The proof will be by (strong) induction on length. We have two base cases: one when the sequence is empty and one when it has one element. On the empty sequence, it returns $-\infty$ as we stated. On any singleton sequence $\langle x \rangle$, the MCSS is x, for which

$$\max\left\{\sum_{k=i}^{j} s_k \;:\; 1 \le i \le 1, 1 \le j \le 1\right\} = \sum_{k=1}^{1} s_k = s_1 = x\,.$$

For the inductive step, let s be a sequence of length $n > 1$, and assume inductively that for any sequence s' of length $n' < n$, $\texttt{mcss}(s')$ correctly computes the maximum contiguous subsequence sum. Now consider the sequence s and let L and R denote the left and right subsequences resulted from dividing s in half (i.e., `NODE(L, R) = showt s`). Furthermore, let $s_{i..j}$ be any contiguous subsequence of s that has the largest sum, and this value is v. Note that the proof has to account for the fact there might be many other subsequences with equal sum. Every contiguous subsequence must start somewhere and end after it. We consider the following 3 possibilities corresponding to how the sequence $s_{i..j}$ lies with respect to L and R:

- If the sequence $s_{i..j}$ starts in L and ends R then its sum equals its part in L (a suffix of L) and its part in R (a prefix of R). If we take the maximum of all suffixes in R and prefixes in L and add them this must equal the maximum of all contiguous sequences bridging the two since $\max\{a + b : a \in A, b \in B\}\} = \max\{a \in A\} + \max\{b \in B\}$. By assumption this equals the sum of $s_{i..j}$ which is v. Furthermore by induction m_L and m_R are sums of other subsequences so they cannot be any larger than v and hence $\max\{m_L, m_R, m_A\} = v$.

- If $s_{i..j}$ lies entirely in L, then it follows from our inductive hypothesis that $m_L = v$. Furthermore m_R and m_A correspond to the maximum sum of other subsequences which cannot be larger than v so again $\max\{m_L, m_R, m_A\} = v$.

- Similarly, if $s_{i..j}$ lies entirely in R, then it follows from our inductive hypothesis that $m_r = \max\{m_L, m_R, m_A\} = v$.

We conclude that in all cases, we return $\max\{m_L, m_R, m_A\} = v$, as claimed. □

Solving these recurrences: Using the definition of big-O, we know that

$$W(n) \quad \leq \quad 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants.

We have solved this recurrence using the tree method. We can also arrive at the same answer by mathematical induction. If you want to go via this route (and you don't know the answer a priori), you'll need to guess the answer first and check it. This is often called the "substitution method." Since this technique relies on guessing an answer, you can sometimes fool yourself by giving a false proof. The following are some tips:

1. Spell out the constants. Do not use big-O—we need to be precise about constants, so big-O makes it super easy to fool ourselves.

2. Be careful that the induction goes in the right direction.

3. Add additional lower-order terms, if necessary, to make the induction go through.

Let's now redo the recurrences above using this method. Specifically, we'll prove the following theorem using (strong) induction on n.

Theorem 3.3. *Let a constant $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then we can find constants κ_1 and κ_2 such that*

$$W(n) \leq \kappa_1 \cdot n \log n + \kappa_2.$$

Proof. Let $\kappa_1 = 2k$ and $\kappa_2 = k$. For the base case ($n = 1$), we check that $W(1) = k \leq \kappa_2$. For the inductive step ($n > 1$), we assume that

$$W(n/2) \leq \kappa_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + \kappa_2,$$

And we'll show that $W(n) \leq \kappa_1 \cdot n \log n + \kappa_2$. To show this, we substitute an upper bound for $W(n/2)$ from our assumption into the recurrence, yielding

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot n \\ &\leq 2(\kappa_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + \kappa_2) + k \cdot n \\ &= \kappa_1 n(\log n - 1) + 2\kappa_2 + k \cdot n \\ &= \kappa_1 n \log n + \kappa_2 + (k \cdot n + \kappa_2 - \kappa_1 \cdot n) \\ &\leq \kappa_1 n \log n + \kappa_2, \end{aligned}$$

where the final step follows because $k \cdot n + \kappa_2 - \kappa_1 \cdot n \leq 0$ as long as $n > 1$. □

Is it possible to do better than $O(n \log n)$ work using divide and conquer?

3.3 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* `(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*

$$(\max(s_1 + p_2, m_1, m_2), \max(p_1, t_1 + p_2), \max(s_1 + t_2, s_2), t_1 + t_2)$$

This gives the following pseudocode:

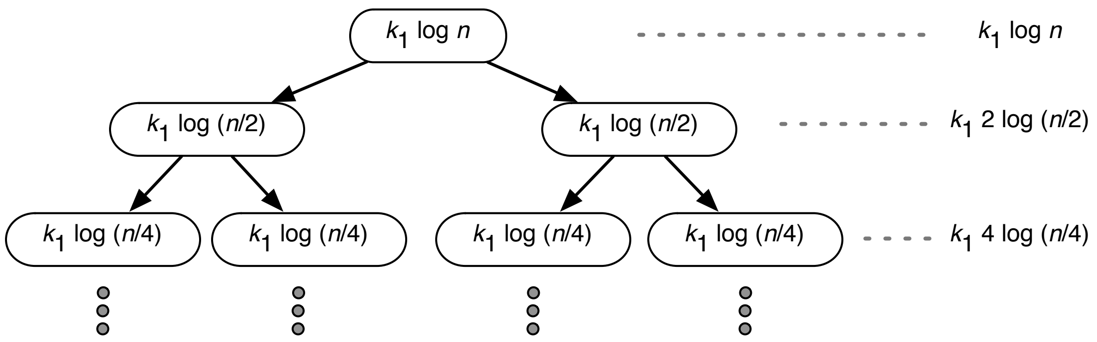
```
fun mcss(a) =
let
  fun mcss'(a)
    case (showt a)
      of EMPTY = (−∞, −∞, −∞, 0)
       | ELT(x) = (x, x, x, x)
       | NODE(L,R) =
          let
            val ((m1, p1, s1, t1), (m2, p2, s2, t2)) = (mcss(L) || mcss(R))
          in
            (max(s1 + p2, m1, m2), max(p1, t1 + p2), max(s1 + t2, s2), t1 + t2)
          end
  val (m, p, s, t) = mcss'(a)
in m end
```

You should verify the base cases are doing the right thing. Also you can see the SML code for this algorithm using SML records at the end of these notes.

Cost Analysis. Assuming `showt` takes $O(\log n)$ work and span, we have the recurrences

$$\begin{aligned} W(n) &= 2W(n/2) + O(\log n) \\ S(n) &= S(n/2) + O(\log n) \end{aligned}$$

These are similar recurrences to what you have in Homework 1. Note that the span is the same as before, so we'll focus on analyzing the work. Using the tree method, we have



Therefore, the total work is upper-bounded by

$$W(n) \le \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i)$$

It is not so obvious what this sum evaluates to. The substitution method seems to be more convenient. We'll make a guess that $W(n) \le \kappa_1 n - \kappa_2 \log n - k_3$. More formally, we'll prove the following theorem:

Theorem 3.4. *Let $k > 0$ be given. If $W(n) \le 2W(n/2) + k \cdot \log n$ for $n > 1$ and $W(n) \le k$ for $n \le 1$, then we can find constants κ_1, κ_2, and κ_3 such that*

$$W(n) \;\le\; \kappa_1 \cdot n - \kappa_2 \cdot \log n - \kappa_3.$$

Proof. Let $\kappa_1 = 3k$, $\kappa_2 = k$, $\kappa_3 = 2k$. We begin with the base case. Clearly, $W(1) = k \le \kappa_1 - \kappa_3 = 3k - 2k = k$. For the inductive step, we substitute the inductive hypothesis into the recurrence and obtain

$$\begin{aligned} W(n) &\le 2W(n/2) + k \cdot \log n \\ &\le 2(\kappa_1 \tfrac{n}{2} - \kappa_2 \log(n/2) - \kappa_3) + k \cdot \log n \\ &= \kappa_1 n - 2\kappa_2(\log n - 1) - 2\kappa_3 + k \cdot \log n \\ &= (\kappa_1 n - \kappa_2 \log n - \kappa_3) + (k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) \\ &\le \kappa_1 n - \kappa_2 \log n - \kappa_3, \end{aligned}$$

where the final step uses the fact that $(k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) = (k \log n - k \log n + 2k - 2k) = 0 \le 0$ by our choice of κ's. $\square$

Finishing the tree method. It is possible to solve the recurrence directly by evaluating the sum we established using the tree method. We didn't cover this in lecture, but for the curious, here's how you can "tame" it.

$$\begin{aligned}
W(n) &\leq \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i) \\
&= \sum_{i=0}^{\log n} k_1 \left(2^i \log n - i \cdot 2^i\right) \\
&= k_1 \left(\sum_{i=0}^{\log n} 2^i\right) \log n - k_1 \sum_{i=0}^{\log n} i \cdot 2^i \\
&= k_1(2n-1)\log n - k_1 \sum_{i=0}^{\log n} i \cdot 2^i.
\end{aligned}$$

We're left with evaluating $s = \sum_{i=0}^{\log n} i \cdot 2^i$. Observe that if we multiply s by 2, we have

$$2s = \sum_{i=0}^{\log n} i \cdot 2^{i+1} = \sum_{i=1}^{1+\log n} (i-1)2^i,$$

so then

$$\begin{aligned}
s &= 2s - s = \sum_{i=1}^{1+\log n} (i-1)2^i - \sum_{i=0}^{\log n} i \cdot 2^i \\
&= \big((1+\log n) - 1\big)\, 2^{1+\log n} - \sum_{i=1}^{\log n} 2^i \\
&= 2n\log n - (2n-2).
\end{aligned}$$

Substituting this back into the expression we derived earlier, we have $W(n) \leq k_1(2n-1)\log n - 2k_1(n\log n - n + 1) \in O(n)$ because the $n\log n$ terms cancel.

4 SML Code

The following code for Algorithm 3 for the MCSS problem uses SML records to give names to each of the four fields. When passing around multiple values such records can make your code more clear and less prone to errors than using unnamed tuples. It is easy to forget the order you store things in tuples, and harder for people who are reading your code to know the order. Fields of a record can be extracted using pattern matching, or a single filed can be accessed using `#fieldname record`, as can be seen at the end of the code.

```
functor mcssDivConq(Seq : SEQUENCE) =
struct
  fun MCSS (A) =
  let
    fun MCSS' A =
      case Seq.showt A
      of Seq.ELT(v) => {mcss = v, prefix = v, suffix = v, total = v}
       | Seq.NODE(A1,A2) =>
         let
            val (B1, B2) = Primitives.par(fn () => MCSS'(A1),
                                          fn () => MCSS'(A2))
            val {mcss = M1, prefix = P1, suffix = S1, total = T1} = B1
            val {mcss = M2, prefix = P2, suffix = S2, total = T2} = B2
         in
            {mcss =   Int.max(S1 + P2, Int.max(M1, M2)),
             prefix = Int.max(P1,      T1 + P2),
             suffix = Int.max(S2,      S1 + T2),
             total = T1 + T2}
         end
  in
    case Seq.showt A
    of Seq.EMPTY => NONE
     |   _       => SOME(#mcss(MCSS' A))
  end
end
```