Lecture 4 — Divide and Conquer

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Margaret Reid-Miller — 24 January 2013

Material in this lecture:
- Maximum Contiguous Subsequence Sum (MCSS) problem with two different divide-and-conquer solutions.
- Solving recurrences using substitution
- Euclidean Traveling Salesperson Problem using divide and conquer.

1 Example I: The Maximum Contiguous Subsequence Sum Problem

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

Definition 1.1 (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

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

(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.

1.1 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

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

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 $O(n^2)$ combinations. Since max reduce for this step 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.

1.2 Algorithm 2: Divide And Conquer — Version 1.0

In coming up with a divide-and-conquer 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:

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

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 answer is incorrect, unfortunately. We need a better understanding of the problem to devise a correct combine step. Notice that the 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 solved 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:

[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 $O(\log n)$

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

Is this algorithm correct? Does it always find the maximum contiguous subsequence sum? Before we show a proof of correctness, what do we consider a proof that an algorithm is correct.

Proofs of correctness. 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. You would prove your ML code was correct line by line. Although proving you code is correct is still important, in this class we will step up a level of abstraction and prove that the algorithms are correct. We still expect your proof to be rigorous. But we are more interested in seeing the critical steps highlighted and the standard or obvious steps 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 1.2. *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 $n' < n$, 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 possibility that there may 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. □

Cost analysis: What is the work and span of this algorithm? We will soon show you how to compute the max prefix and suffix sums in parallel. For now, we will 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. This yields the following recurrences:

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

Using the definition of big-O, we know that

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

where k_1 and k_2 are constants.

We have solved this recurrence using the recursion 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 inequalities always go 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 1.3. *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?

1.3 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. What information from the subproblems would be help to make the combine step take constant work instead of linear work?

In the design of our previous algorithm, if we knew max suffix sum and max prefix sums of the subproblems, we could produce the max subsequence sum in constant time. The expensive part was 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 max prefix and suffix sums in addition to the max subsequence sum. Is this enough? As we showed above, these are sufficient to find the final mcss. But can we find the max prefix and max suffix that we also need to return? No, as the max prefix or max suffix could span across the divide point. For example, the max prefix could either be the a prefix of the left subproblem as before, or it could be all of the left subproblem and a prefix of the right subproblem. Therefore, we also need the overall sum of each subproblem.

So, we need to return a total of four 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:

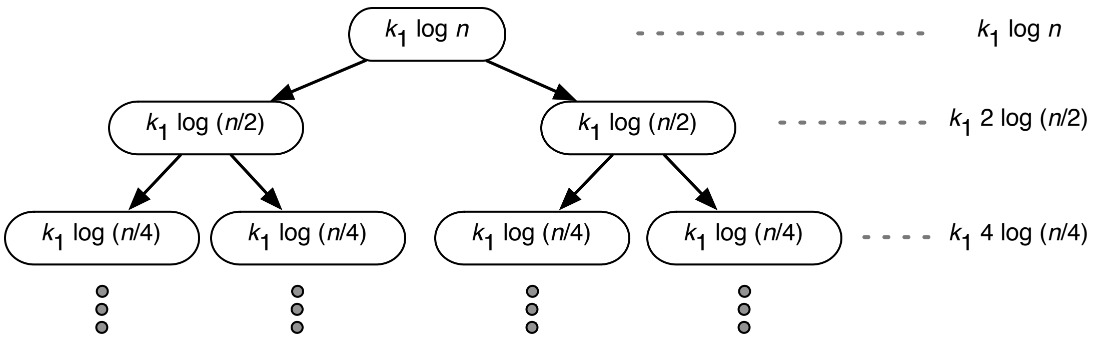
```
function mcss($a$) =
let
  function mcss'($a$)
    case (showt $a$)
      of EMPTY = $(-\infty, -\infty, -\infty, 0)$
       | ELT($x$) = $(x, x, x, x)$
       | NODE($L$,$R$) =
         let
            $((m_1, p_1, s_1, t_1), (m_2, p_2, s_2, t_2))$ = (mcss($L$) $\|$ mcss($R$))
         in
           $(\max(s_1 + p_2, m_1, m_2),\ \max(p_1, t_1 + p_2),\ \max(s_1 + t_2, s_2),\ t_1 + t_2)$
         end
   $(m, p, s, t) = mcss'(a)$
in $m$ end
```

You should verify the base cases are doing the right thing. The SML code for this algorithm is at the end of these notes.

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

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

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

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

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

Theorem 1.4. *Let $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot \log n$ for $n > 1$ and $W(n) \leq k$ for $n \leq 1$, then we can find constants κ_1, κ_2, and κ_3 such that*

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

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

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

where the final step uses the fact that $(k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) = (k \log n - k \log n + 2k - 2k) = 0 \leq 0$ 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

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

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.

Example II: The Euclidean Traveling Salesperson Problem

We'll now turn to another example of divide and conquer. In this example, we will apply it to devise a heuristic method for an **NP**-hard problem. The problem we're concerned with is a variant of the traveling salesperson problem (TSP) from Lecture 1. This variant is known as the Euclidean traveling salesperson (eTSP) problem because in this problem, the points (aka. cities, nodes, and vertices) lie in a Euclidean space and the distance measure is the Euclidean measure. More specifically, we're interested in the planar version of the eTSP problem, defined as follows:

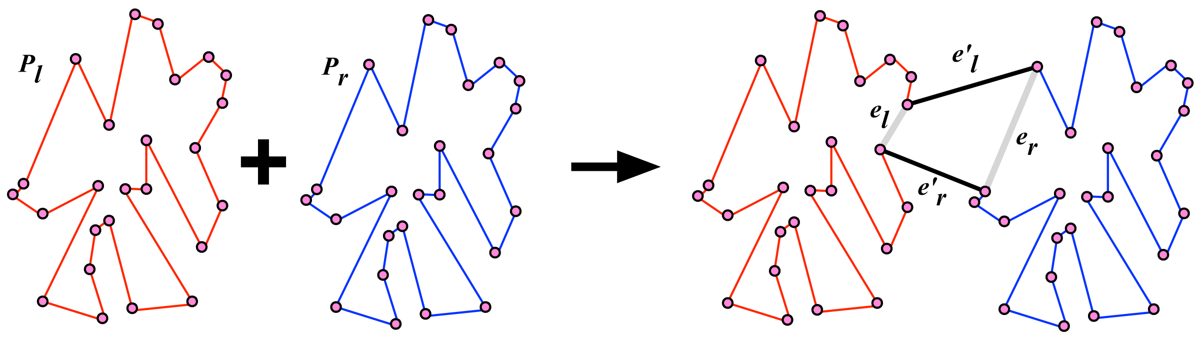
Definition 1.5 (The Planar Euclidean Traveling Salesperson Problem)**.** Given a set of points P in the 2-d plane, the *planar Euclidean traveling salesperson* (eTSP) problem is to find a tour of minimum total distance that visits all points in P exactly once, where the distance between points is the Euclidean (i.e. ℓ_2) distance.

Not counting bridges, this is the problem we would want to solve to find a minimum length route visiting your favorite places in Pittsburgh. As with the TSP, it is **NP**-hard, but this problem is easier[2] to approximate.

Here is a heuristic divide-and-conquer algorithms that does quite well in practice. In a few weeks, we will see another algorithm based on Minimum Spanning Trees (MST) that gives a constant-approximation guarantee. This divide-and-conquer algorithm is more interesting than the ones we have done so far because it does work both before and after the recursive calls. Also, as we will see, the recurrence it generates is root dominated.

The basic idea is to split the points by a cut in the plane, solve the TSP on the two halves, and then somehow merge the solutions. For the cut, we can pick a cut that is orthogonal to the coordinate lines. In particular, we can find in which of the two dimensions the points have a larger spread, and then find the median point along that dimension. We'll split just below that point.

To merge the solutions we join the two cycles by making swapping a pair of edges.



To choose which swap to make, we consider all pairs of edges of the recursive solutions consisting of one edge $e_\ell = (u_\ell, v_\ell)$ from the left and one edge $e_r = (u_r, v_r)$ from the right and determine which

[2]Unlike the TSP problem, which only has constant approximations, it is known how to approximate this problem to an arbitrary but fixed constant accuracy ε in polynomial time (the exponent of n has $1/\varepsilon$ dependency). That is, such an algorithm is capable of producing a solution that has length at most $(1 + \varepsilon)$ times the length of the best tour.

pair minimizes the increase in the following cost:

$$\texttt{swapCost}((u_\ell, v_\ell),(u_r, v_r)) = \left\|u_\ell - v_r\right\| + \left\|u_r - v_\ell\right\| - \left\|u_\ell - v_\ell\right\| - \left\|u_r - v_r\right\|$$

where $\|u - v\|$ is the Euclidean distance between points u and v.

Here is the pseudocode for the algorithm

```
function eTSP($P$) =
  case ($|P|$)
    of 0,1 $\Rightarrow$ raise $TooSmall$
     | 2 $\Rightarrow$ $\{(P[0],P[1]),(P[1],P[0])\}$
     | $n$ $\Rightarrow$ let
          $(P_\ell, P_r)$ = splitLongestDim($P$)
          (L,R) = ( eTSP($P_\ell$) || eTSP($P_r$) )
          $(c,(e,e'))$ = $\texttt{minVal}_{\texttt{first}}$ $\{(\texttt{swapCost}(e,e'),(e,e')) : e \in L, e' \in R\}$
       in
         swapEdges(append($L,R$), $e, e'$)
       end
```

The function $\texttt{minVal}_{\texttt{first}}$ 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' in E and swaps the endpoints. As there are two ways to swap, it picks the cheaper one.

Cost analysis Now let's analyze the cost of this algorithm in terms of work and span. We have

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

We have already seen the recurrence $S(n) = S(n/2) + O(\log n)$, which solves to $O(\log^2 n)$. Here we'll focus on solving the work recurrence.

In anticipation of recurrences that you'll encounter later in class, we'll attempt to solve a more general form of recurrences. Let $\varepsilon > 0$ be a constant. We'll solve the recurrence

$$W(n) = 2W(n/2) + k \cdot n^{1+\varepsilon}$$

by the substitution method.

Theorem 1.6. *Let $\varepsilon > 0$. If $W(n) \le 2W(n/2) + k \cdot n^{1+\varepsilon}$ for $n > 1$ and $W(1) \le k$ for $n \le 1$, then for some constant κ,*

$$W(n) \le \kappa \cdot n^{1+\varepsilon}.$$

Proof. Let $\kappa = \frac{1}{1-1/2^\varepsilon} \cdot k$. The base case is easy: $W(1) = k \le \kappa_1$ as $\frac{1}{1-1/2^\varepsilon} \ge 1$. For the inductive step, we substitute the inductive hypothesis into the recurrence and obtain

$$\begin{aligned} W(n) &\le 2W(n/2) + k \cdot n^{1+\varepsilon} \\ &\le 2\kappa \left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} \\ &= \kappa \cdot n^{1+\varepsilon} + \left(2\kappa \left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon}\right) \\ &\le \kappa \cdot n^{1+\varepsilon}, \end{aligned}$$

where in the final step, we argued that

$$\begin{aligned} 2\kappa\left(\frac{n}{2}\right)^{1+\varepsilon} + k\cdot n^{1+\varepsilon} - \kappa\cdot n^{1+\varepsilon} &= \kappa\cdot 2^{-\varepsilon}\cdot n^{1+\varepsilon} + k\cdot n^{1+\varepsilon} - \kappa\cdot n^{1+\varepsilon} \\ &= \kappa\cdot 2^{-\varepsilon}\cdot n^{1+\varepsilon} + (1-2^{-\varepsilon})\kappa\cdot n^{1+\varepsilon} - \kappa\cdot n^{1+\varepsilon} \\ &\le 0. \end{aligned}$$

□

Solving the recurrence directly. Alternatively, we could use the tree method and evaluate the sum directly. As argued before, the recursion tree here has depth $\log n$ and at level i (again, the root is at level 0), we have 2^i nodes, each costing $k\cdot(n/2^i)^{1+\varepsilon}$. Thus, the total cost is

$$\begin{aligned} \sum_{i=0}^{\log n} k\cdot 2^i\cdot\left(\frac{n}{2^i}\right)^{1+\varepsilon} &= k\cdot n^{1+\varepsilon}\cdot\sum_{i=0}^{\log n} 2^{-i\cdot\varepsilon} \\ &\le k\cdot n^{1+\varepsilon}\cdot\sum_{i=0}^{\infty} 2^{-i\cdot\varepsilon}. \end{aligned}$$

But the infinite sum $\sum_{i=0}^{\infty} 2^{-i\cdot\varepsilon}$ is at most $\frac{1}{1-1/2^{\varepsilon}}$. Hence, we conclude $W(n)\in O(n^{1+\varepsilon})$.

2 SML Code

Note that 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. Unlike tuples, you can extract only the fields that you need by including ellipse (...) at the end of the pattern. As with tuples you can accessed a single field 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
```