Lecture 3 — Maximum Contiguous Subsequence Sum and The Substitution Method

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

Lectured by Kanat Tangwongsan — January 24, 2012

Material in this lecture: The main theme of today's lecture is *divide and conquer*.

- Maximum contiguous subsequence sum (MCSS)
- Proof of correctness
- Solving recurrences using the substitution method.

1 Maximum Contiguous Subsequence Sum Problem

Consider the maximum contiguous subsequence sum (MCSS) problem, defined as follows:

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 maximizies the sum. In this particular example, we can check that $\mathsf{mcss}(s) = 4$, achieved by taking the subsequence $\langle 2, -1, 3 \rangle$.

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 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 $\langle s_i, s_{i+1}, \ldots, s_j \rangle$.

For each subsequence $i..j$, we can compute its sum by applying a plus `reduce`. This does $O(j - i)$ work and $O(\log(j - i))$ span. Furthermore, all the subsequences can be examined independently in parallel (using, e.g., `tabulate`). This leads the following bounds:

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

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

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.

1.2 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 split a sequence? Let's split the sequence in half, recursively solve the problem on both halves and see what we can do. For example, imagine we split 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 split 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 $\max(\text{mcss}(L), \text{mcsss}(R))$. This is unfortunately incorrect. 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 split 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 cross the split 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 $O(\log n)$

```
fun mcss(s) =
  case (showt s)
    of EMPTY = $-\infty$
     | ELT(x) = $\max(x, 0)$
     | NODE(L,R) =
         let val $(m_L, m_R)$ = par(mcss(L) || mcss(R))
             val $best\_across$ = bestAcross(L,R)
         in $\max\{m_L, m_R, best\_across\}$
         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 splitting 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 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. For the base case, we consider the following cases: on the empty sequence, it returns $-\infty$—our symbol for undefined—as expected, and on any singleton sequence $\langle x \rangle$, the MCSS is $\max(0, x)$, which is what it produces.

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 spliting s in half (i.e., `NODE(L, R) = showt s`). Furthermore, let $s_{i..j}$ be any subsequence of s that has the largest sum among all subsequences of s. Say $s_{i..j}$ sums to v, so

$$\max\{m_L, m_R, \texttt{best\_across}\} \leq v \tag{1}$$

since these are subsequences of s.

We consider the following 3 possibilies corresponding to how the sequence $s_{i..j}$ lies with respect to L and R:

- If $s_{i..j}$ lies entirely in L, then it follows from our inductive hypothesis that `mcss(L)` $= v$, which means the answer we return $\max\{m_L, m_R,$ `best_across`$\} \geq m_L = v$. This, together with (1), implies that the answer we return is exactly v.
- Similarly, if $s_{i..j}$ lies entirely in R, then it follows from our inductive hypothesis that `mcss(R)` $=$ v, which means the answer we return $\max\{m_L, m_R,$ `best_across`$\} \geq m_R = v$. This, together with (1), implies that the answer we return is exactly v
- Otherwise, the sequence $s_{i..j}$ starts in L and ends R. We prove a separate lemma that `best_across` gives us the maximum contiguous subsequence sum that begins in L and ends in R (proof needed but omitted for these notes). This lemma then tells us that `best_across` $= v$, so the answer that we produce $\max\{m_L, m_R,$ `best_across`$\} \geq$ `best_across` $= v$, as desired.

We conclude that in all cases, we return $\max\{m_L, m_R,$ `best_across`$\} = v$, 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 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 Digression: A Useful Trick — The Brick Method

As we have seen from the previous lecture, 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—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 asympotic 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:

Root Dominated	***Leaves Dominated***	***Balanced***
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, $\mathsf{cost}_{i+1} \leq \rho \cdot \mathsf{cost}_i$	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.
+++++++++ ++++++ ++++ ++	++ ++++ ++++++ ++++++++	++++++++ ++++++++ ++++++++ ++++++++
Implication: $O(\mathsf{cost}_0)$.	*Implication:* $O(\mathsf{cost}_d)$, where d is the depth.	*Implication:* $O(d \cdot \max_i \mathsf{cost}_i)$

1.4 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*

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

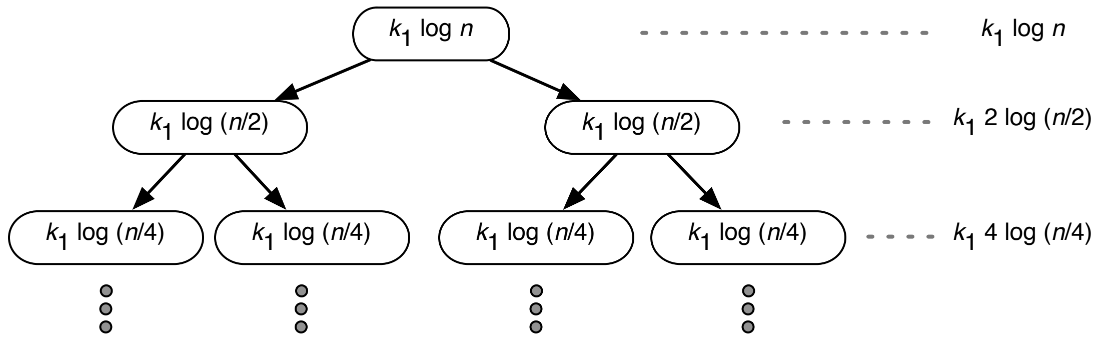
This gives the following ML code:

```
fun MCSS (A) =
let
   fun MCSS' (A) =
     case showt(A) of
        EMPTY => {mcss=0, prefix=0, suffix=0, total=0}
      | ELT(v) =>
         {mcss=Int.max(v,0), prefix=Int.max(v,0), suffix=Int.max(v,0), total=v}
      | NODE(A1,A2) =>
        let
           val (B1, B2) = (MCSS'(A1), 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
    val {mcss = B, ...} = MCSS'(A)
in
  B
end;
```

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

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

It is not so obvious what this sum evaluates to. The substitution method seems to be more convenient. We'll make a guess that $W(n) \leq \kappa_1 n - \kappa_2 \log n - k_3$. 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 $\kappa_1 = 1$, $\kappa_2 = 2k$, $\kappa_3 = k$. We begin with the base case. Clearly, $W(1) = k \leq \kappa_1 - \kappa_3$. For the inductive step, we substitute the inductive hypothesis into the recurrence and obtain

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

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

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.

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

We're left with evaluating $s = \sum_{i=0}^{\log n} i \cdot 2^i$. Observe that if we mulitply 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 $W(n) \le k_1(2n-1)\log n - 2k_1(n\log n - n + 1) \in O(n)$ because the $n\log n$ terms cancel.

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 1.5. *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 $W(n) \in O(n^{1+\varepsilon})$.