Lecture 3 — More Divide-and-Conquer and Costs

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 6, 2011

1 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 class, but this 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.

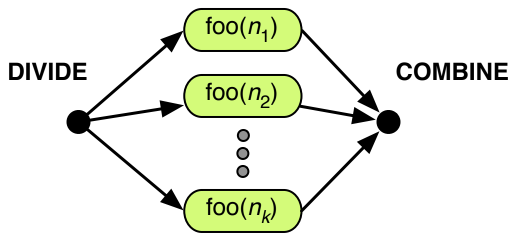
In most divide-and-conquer algorithms you have encountered so far, the subproblems are occurrences of the problem you are solving (e.g. recursive sorting in merge sort). This is not always the case. Often, you'll need more information from the subproblems to properly combine results of the subproblems. In this case, you'll need to strengthen the problem, much in the same way that you strengthen an inductive hypothesis when doing an inductive proof. We will go through some examples in this class where problem strengthening is necessary. But you have seen some such examples already from Recitation 1 and your Homework 1.

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, we 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 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 as the following simple recurrences: If the 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. 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.

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.

2 Example I: Maximum Contiguous Subsequence Sum Problem

The first example we're going to look at in this lecture is the maximum contiguous subsequence sum (MCSS) problem. This can be defined as follows.

Definition 2.1 (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).

2.1 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:

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

Note that these bounds didn't include the cost of the max reduce taken over all these sequences. This max reduce has $O(n^2)$ work and $O(\log n)$ span[1]; therefore, the cost of max reduce is subsumed by the other costs analyzed above. Overall, this is a $O(n^3)$-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.

Exercise 1. *Can you improve the work of the naïve algorithm to $O(n^2)$?*

2.2 Algorithm 2: Divide And Conquer — Version 1.0

We'll design a divide-and-conquer algorithm for this problem. An important first step in coming up with such an algorithm 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:

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

How would we use this to create a divide-and-conquer algorithm? So far, we have suggested a simple way to split the input sequence, so now the question to answer is, what are the possibilities for how the maximum subsequence sum is formed? 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.

What information do we need to keep track of to handle this case? If we take the largest sum of a suffix on the left and the largest sum of a prefix on the right, then we will get the largest sum that goes between the two. Then we just take a max over the three possibilities, in left (56), in right (17), or between.

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 $O(n)$ 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

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

where k is a positive constant.

[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 $O(\log n)$

Solving these recurrences: We'll now focus on solving these recurrences, resorting to the tree method, which you have seen in 15-122 and 15-251.

First, let's think through what it means when we write $O(f(n))$ in an expression (e.g., when we write $2W(n/2) + O(n)$ 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 such that for all $n \geq 1$, and k_2 such that*

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

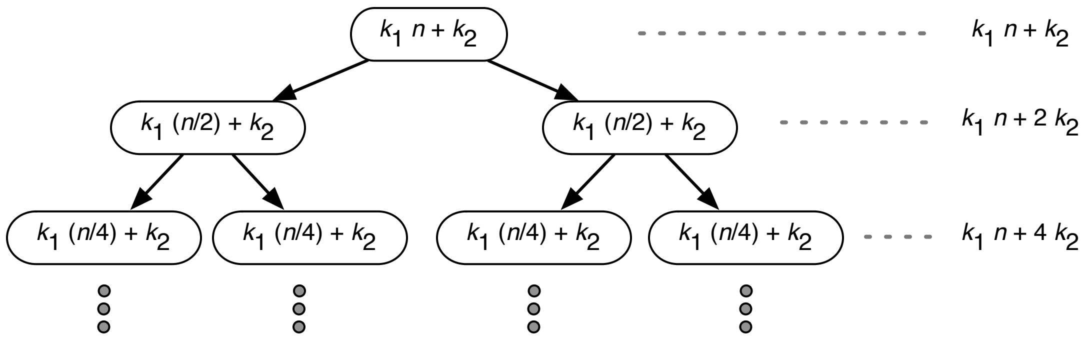
where, for example, we can take $k_1 = c$ and $k_2 = \sum_{i=1}^{N_0} g(i)$, assuming f and g are non-negative functions.

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 go about solving this recurrence using 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 work, 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:



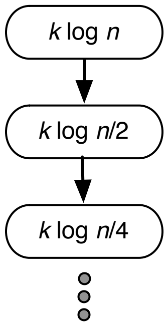
It can be seen that level i (the root is level $i = 0$) contains 2^i nodes, each costing at most $k_1(n/2^i) + k_2$. Thus, the total cost in level i is at most

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

How many levels are there in this tree? Since we keep halving the input size, the number of levels is bounded by $1 + \log n$. Hence, we have

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

As for span, by unfolding the recurrence, we have the following recursion tree:



We conclude that the span is

$$S(n) \;=\; \sum_{i=0}^{\log n} k\log(n/2^i) \;=\; \sum_{i=0}^{\log n} k(\log n - i) \;\in\; O(\log^2 n).$$

Guess and verify by induction: Alternatively, we can also arrive at the same answer by mathematical induction. If you want to go via this route, 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 to avoid common mistakes:

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 2.2. *If* $W(n) \le 2W(n/2) + k\cdot n$ *for* $n > 1$ *and* $W(1) \le k$ *for* $n \le 1$*, then for some constants* κ_1 *and* κ_2*,*

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

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

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

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

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

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?

2.3 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 ML code:

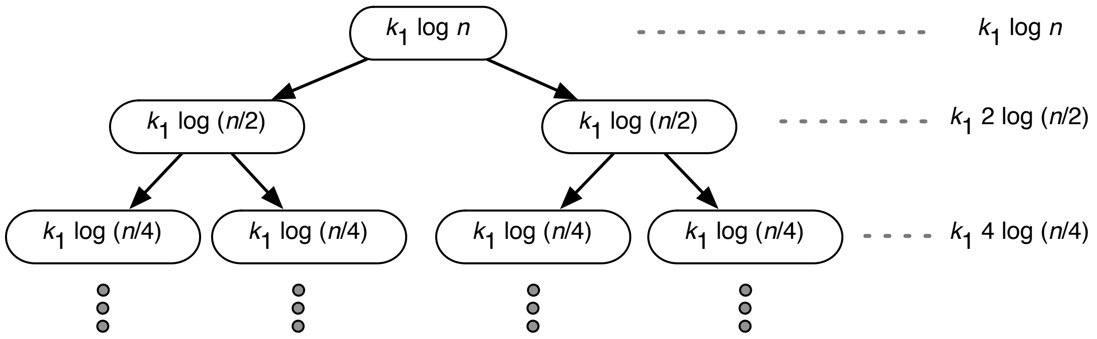
```
fun MCSS (A) =
let
   fun MCSS' (A) =
     case showt(A) of
        EMPTY => {mcss=0, prefix=0, suffix=0, total=0}
      | ELT(v) =>
         {mcss=Int.max(v,0), prefix=Int.max(v,0), suffix=Int.max(v,0), total=v}
      | NODE(A1,A2) =>
        let
           val (B1, B2) = (MCSS'(A1), MCSS'(A2))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

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

It is not so obvious what this sum equals. The substitution method seems to be more convenient. We'll make a guess that $W(n) \leq \kappa_1 n - \kappa_2 \log n - \kappa_3$. More formally, we'll prove the following theorem:

Theorem 2.3. *If* $W(n) \leq 2W(n/2) + k \cdot \log n$ *for* $n > 1$ *and* $W(1) \leq k$ *for* $n \leq 1$, *then for some constants* κ_1 *and* κ_2,

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

Proof. Let $\kappa_1 = 2k$, $\kappa_2 = \kappa_3 = k$. 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

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

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

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

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

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

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.

3 Example II: The Euclidean Traveling Salesperson Problem

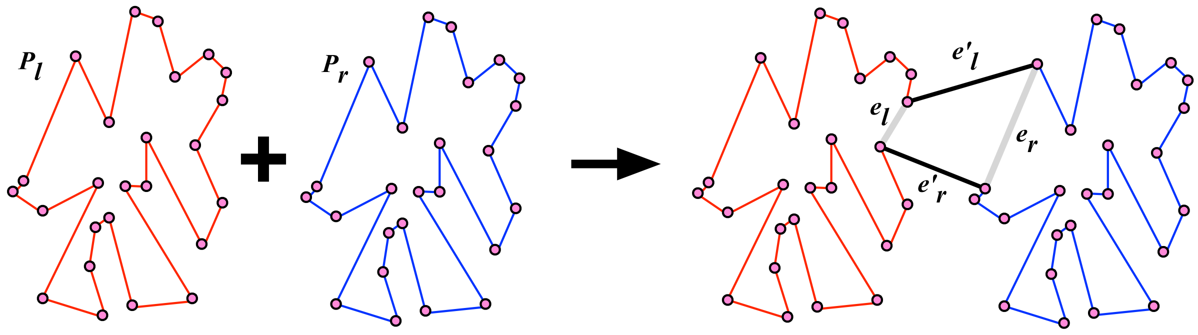
We'll now turn to another example of divide and conquer. In this example, we will apply it to devise a heuristic method for an **NP**-hard problem. The problem we're concerned with is a variant of the traveling salesperson problem (TSP) from Lecture 1. This variant is known as the Euclidean traveling 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 3.1 (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.

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.

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 an edge swap



To choose which edge swap to make, we consider all pairs of edges each from one side and determine which one minimizes the increase in cost: swapCost$((u_\ell, v_\ell), (u_r, v_r)) = \|u_\ell - v_r\| + \|u_r - v_\ell\| - \|u_\ell - v_\ell\| - \|u_r - v_r\|$, where $\|u - v\|$ is the Euclidean distance between points u and v.

Here is the pseudocode for the algorithm

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

You can find the full code online. Now let's analyze the cost of this algorithm in terms of work and

[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.

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

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

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

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

where in the final step, we argued that

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

□

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

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

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