Lecture 3 — Algorithmic Techniques and Analysis

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

Lectured by Margaret Reid-Miller — 22 January 2013

Material in this lecture:
- Algorithmic techniques
- Asymptotic Analysis
- Divide and Conquer Recurrences

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 are solutions 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 Umut A. Acar, 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 subproblems 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 $k > 1$ independent 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. It is important than the subproblems are independent; this makes the approach amenable to parallelism because independent problems can be solved in parallel.
- *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. Contraction 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. Strengthening 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, since the original is the special case when both these values are zero (no unmatched right or left parentheses). Not only does the modified version tells us more than we, it was necessary to make divide-and-conquer work.

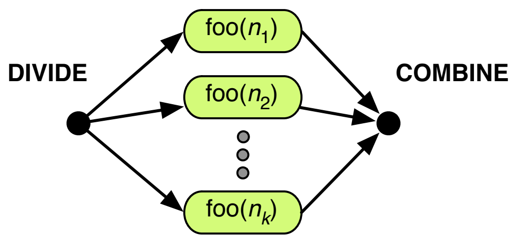
The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction, which makes it easy to show correctness. Often it also makes it easy to determine costs bounds since we can write recurrences that match the inductive structure. The general form of divide-and-conquer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



Since the subproblems can be solved independently (by assumption), 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 $n \geq 1$,*

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

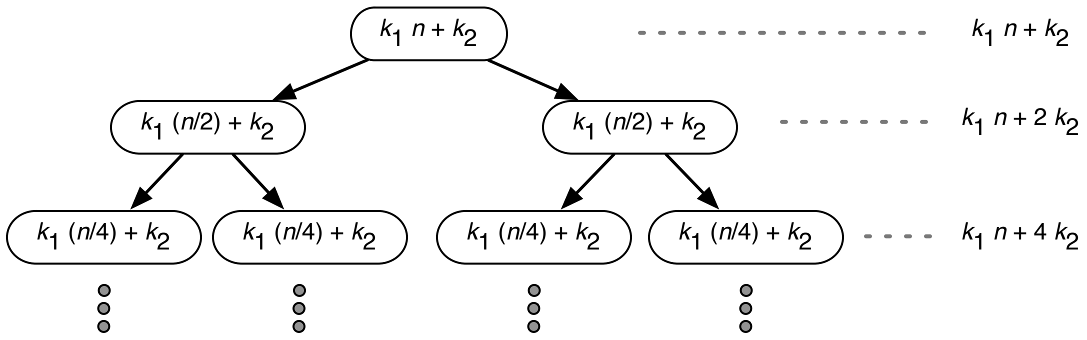
where, for example, we can take $k_1 = c$ and $k_2 = \sum_{i=1}^{N_0} |g(i)|$.

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 denote 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.

2.2 Substitution method:

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.

Besides using the recursion tree method, 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 2.1. *Let a constant $k > 0$ be given. If $W(n) \le 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \le k$ for $n \le 1$, then we can find constants κ_1 and κ_2 such that*

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

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 $k \cdot n + \kappa_2 - \kappa_1 \cdot n \le 0$ as long as $n > 1$. □