Lecture 3 — Algorithmic Techniques and Analysis

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

Lectured by Umut Acar — 3 September 2013

Material in this lecture:

- Algorithmic techniques
- Asymptotic Analysis
- Divide and Conquer Recurrences

1 Algorithmic Techniques

One of the things that students in the previous semesters found most difficult in this class was coming up with their work algorithms. One difficulty with this is that one does not know where to start.

Question 1.1. *Have you designed algorithms before. If so what approaches have you found helpful?*

Given an algorithmic problem, where do you even start? It turns out that most of the algorithms follow several well-known techniques. For example, when solving the shortest superstring (SS) problem we already mentioned three techniques: brute force, reducing one problem to another, and the greedy approach. In this lecture, we will go over these techniques, which are key to both sequential and parallel algorithms, and focus on one of them, divide and conquer, which turns out to be particularly useful for parallel algorithm design. We will also talk about asymptotic cost analysis and how to analyze divide-and-conquer algorithms.

Remark 1.2. To become good at designing algorithms, we would recommend gaining as much experience as possible by both formulating problems and solving them by applying the techniques that we will discover in this lecture.

Brute Force: The brute force approach involves trying all possible solutions and picking the best or one of the satisfying solution (typically the first encountered). 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. In some problems, it suffices to pick a solution; in such cases, it is not necessary to consider all possible solutions, a brute-force algorithm can return the first encountered. 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.

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

Question 1.3. *Given that they are often inefficient, why do we care about brute-force algorithms?*

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 usually the simplest solution to a problem, but not always.

Question 1.4. *When trying to apply the brute-force technique to solve a problem, where would you start first?*

Since brute-force requires enumerating all possible results and checking them, it is often helpful to start by identifying the structure of the expected results (all strings, all natural numbers between 0 and n) and a way to assign values to them so that you can pick the best.

Reducing to another problem: Sometimes the easiest approach to solving a problem is just reduce it to another problem for which known algorithms exist.

Question 1.5. *You all know about the problem of finding the minimum number in a set. Can you reduce the problem of finding the maximum to this problem?*

Simply invert the signs of all numbers.

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?

Question 1.6. *Can you think of an exercise for becoming better in using this technique?*

When you get more experienced with algorithms you will start recognizing similarities between problems that on the surface seem very different. But one exercise that you might find helpful would be to formulate new algorithmic problems, as this can help in understanding how seemingly different problems may relate.

Inductive techniques: The idea behind inductive techniques is to solve one or more smaller instances of the same problem, typically referred to as *subproblems*, to solve the original problem. The technique most often uses recursion to solve the subproblems. 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

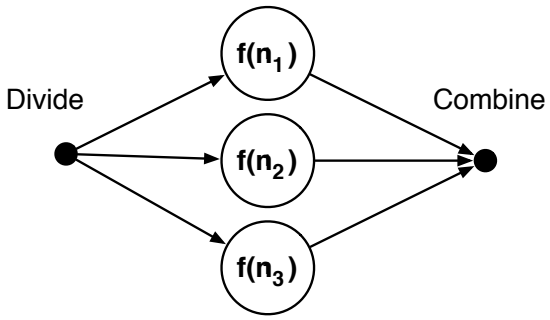


Figure 1: Structure of a divide-and-conquer algorithm illustrated ($k = 1$).

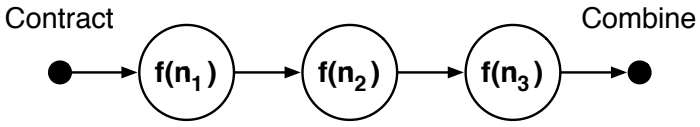


Figure 2: Structure of a contraction algorithm illustrated.

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 or instances (e.g., of size $n/2$), solve the smaller instances sequentially, and then use the results to solve the original problem. Contraction only differs from divide and conquer in that it allows there to be only one subproblem or many dependent subproblems.
- *Dynamic Programming.* Like divide and conquer, dynamic programming divides the problem into smaller subproblems 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.

Randomization: Randomization in is a powerful algorithm design technique that can be applied along with the aforementioned techniques. It often leads to simpler algorithms.

Question 1.7. *Do you know of a randomized and divide-conquer algorithm?*

Randomization is also crucial in parallel algorithm design as it helps un "break" the symmetry in a problem without communication.

Question 1.8. *Have you heard of the term "symmetry breaking"? Can you think of a real-life phenomena that is somewhat amusing and can even be uncomfortable where we engage in randomized symmetry breaking?*

We often perform randomized symmetry breaking when walking in a crowded sidewalk with lots of people coming in our direction. With each person we encounter, we may pick a random direction to turn to and other person responds (or sometimes we do). If we recognize each other late, we may end up in a situation where the randomness becomes more apparent, as we attempt to get past each other's way but make the same (really opposite) decisions. Since we make essentially random decisions, the symmetry is eventually broken.

We will cover a couple of examples of randomized algorithms in this course. Formal cost analysis for many randomized algorithms, however, can (but not always) require knowledge of probability theory beyond the level of this course.

What is a good solution? When you encounter an algorithmic 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.

Question 1.9. *How do you show that your solution is good?*

Ideally you should prove the correctness and efficiency of your algorithm. For example, in exams, we might ask you do this. Even in real life, we would highly recommend making a habit of this as well. It is often too easy to convince yourself that a poor algorithm is correct and efficient.

Question 1.10. *How can you prove correct an algorithm designed by using an inductive technique? Can you think of a proof technique that might be useful?*

Algorithms designed with inductive techniques can be proved correct using (strong) induction.

Question 1.11. *How can you analyze an algorithm designed by using an inductive technique? Can you think of a technique that might be useful?*

We can often express the complexity of inductive algorithms with recursions and solve them to obtain a closed-form solution.

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.

The structure of divide and conquer. 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.

Question 2.1. *Can you identify the base case and the inductive step for the quicksort algorithm working on a set of numbers?*

The base case is when the set being sorted contains only one element. The inductive case partitions the set into two sets by picking a random pivot. This partition is the divide step. The algorithm then recurses on the two subproblems and combines the solution by appending them. In quicksort combine is easy and cheap. Most of the work is done in the divide step.

Question 2.2. *Can you think of different base cases for quicksort that may lead to lower cost?*

It turns out asymptotically inefficient algorithms such as insertion sort have lower constant factors and may be more efficient in practice when solving small problem instances (typically less than 100). We can thus improve the efficiency of quicksort by picking a larger base case and reverting to a different algorithm in the base case.

Question 2.3. *Can you identify the base case and the inductive step for the mergesort algorithm working on a set of numbers? How does it differ from quicksort?*

In the base case, we have two or fewer elements in the set. In the inductive case, we split the set into two equal halves (divide step), solve them recursively, and merge the results (combine step). Mergesort is different from quicksort because the divide step in trivial but combine is more involved—that is where most of the work is done.

As we will see later, we can express divide and conquer algorithms with trivial divide steps with reduce by using higher-order functions.

Exercise 1. *Can you think of different base cases for mergesort that may lead to lower cost? Will it be the same as that for quicksort?*

Strengthening. 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.

Question 2.4. *You have recently seen an instance of strengthening when solving a problem with divide and conquer. Can you think of the problem and how you used strengthening?*

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. In fact, if we do not strengthen the problem, we will not be able to combine the results from the two recursive calls (which tell you only whether two halves are mathed or not) to conclude that the full string is matched, because for example there can be an unmatched open paranthesis on one side that matches a close parenthesis on the other.

Work and Span. 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

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

and the span is

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

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.

3 Asymptotic Complexity

What is $O(f(n))$? Precisely, $O(f(n))$ is the set of all functions that asymptotically dominated by the function $f(n)$. Intuitively this means that these are functions that grow at the same or slower rate than $f(n)$ as n goes to infinity. We write $g(n) \in O(f(n))$ to refer to a function $g(n)$ that is in the set $O(f(n))$. We often use the abusive notation $g(n) = O(f(n))$ as well.

In an expression such as $4W(n/2) + O(n)$, the $O(n)$ refers to some function $g(n) \in O(n)$.

Question 3.1. *Do you remember the definition of $O(\cdot)$?*

For a function $g(n)$, we say that $g(n) \in O(f(n))$ if there exist positive constants n_0 and c such that for all $n \geq n_0$, we have $g(n) \leq c \cdot f(n)$.

If $g(n)$ is a finite function ($g(n)$ in finite for all n), then 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 $k_2 = \sum_{i=1}^{n_0} |g(i)|$.

Remark 3.2. Make sure to become very comfortable with asymptotic analysis. Also its different versions such as the $\Theta(\cdot)$ and $\Omega(\cdot)$.

Exercise 2. *Can you illustrate graphically when $g(n) \in O(f(n))$? Show different examples, to hone your understanding.*

4 Solving recurrences: Tree Method

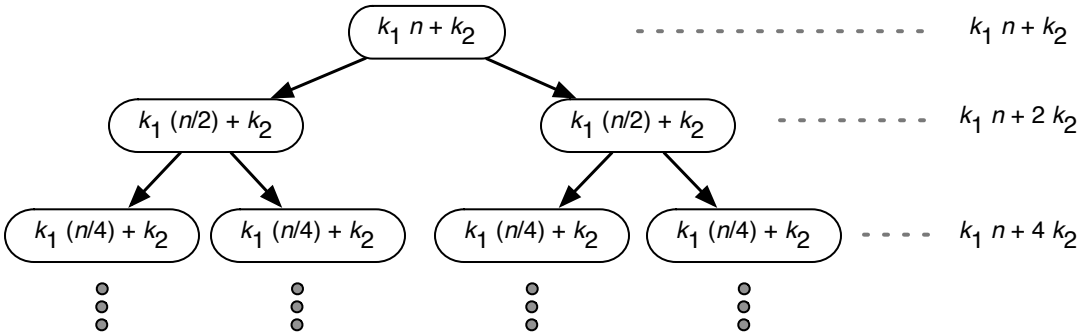
Using the recursion $W(n) = 2W(n/2) + O(n)$, we will review the tree method, which you have seen in 15-122 and 15-251. Our goal is to derive a closed form solution to this recursion.

By the definition of asymptotic complexity, we can establish that

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

where k_1 and k_2 are constants.

The idea of the tree method is to consider the recursion tree of the recurrence 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 $a > 1$,

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

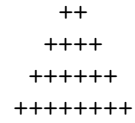
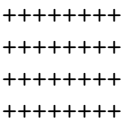
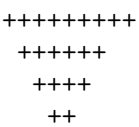
5 Solving Recurrences with a Trick: Brick Method

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.

It turns out that there is a special case in which the analysis becomes simpler. This is whet the costs at each level grow geometrically, shrink geometrically, or stay approximately equal.

By recognizing whether the recurrence conforms with one of these cases, 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.

5.1 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 5.1. *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 $n > 1$. □