CSE347 Class 2

Jeremy Buhler

August 30, 2017

1 Paradigms of Algorithm Development

- We've seen one example of an algorithm, with attendant correctness and efficiency arguments.
- We could keep studying individual algorithms pretty much forever... [perhaps you can name a few of your favorites]
- But this course is not *just* about building a toolbox of standalone methods.
- We need some *structure* – are there general design principles that can be used to obtain efficient algorithms for many different problems?
- In the coming weeks, we're going to look at three such principles:
  1. divide-and-conquer
  2. greedy algorithms
  3. dynamic programming
- Today, we'll get a taste of divide-and-conquer.
- You may have seen this principle used previously in, e.g., merge sorting a list of numbers, or in binary search of a sorted array.

2 Matrix Multiply

Let's think about matrices!

- Consider taking a matrix product $A \times B = C$.
- Assume A is an $n \times p$ matrix, and B is a $p \times m$ matrix.
- Then C has size (wait for it) $n \times m$.
- More specifically, the i, jth entry in C is computed as

$$C(i,j) = \sum_{k=1}^{p} A(i,k) \cdot B(k,j).$$

- So how many arithmetic operations are needed to compute C from A and B?

- There are nm entries in C.
- Each entry of C needs p multiplies and $p-1$ adds.
- Hence, total cost of computation is $\Theta(npm)$ operations.
- If $n = p = m$, we may say that matrix multiply is a $\Theta(n^3)$ algorithm.
- (Compare to only $\Theta(n^2)$ operations to add two $n \times n$ matrices.)

Hrmph. That kind of sucks. Can we do better?

- Suppose for simplicity that n is a power of 2, and A, B have size $n \times n$.
- What if we break A and B into four chunks of size $n/2 \times n/2$?
- (BTW, these chunks are called "minors" of the matrix.)
- We can write A, B, and C as 2×2 matrices of four chunks, like this:

- Now we can express the product $C = A \times B$ in terms of the products on the chunks:

$$\begin{aligned}
C^{11} &= A^{11} \times B^{11} + A^{12} \times B^{21} \\
C^{12} &= A^{11} \times B^{12} + A^{12} \times B^{22} \\
C^{21} &= A^{21} \times B^{11} + A^{22} \times B^{21} \\
C^{22} &= A^{21} \times B^{12} + A^{22} \times B^{22}.
\end{aligned}$$

- This approach uses a total of 4 additions and 8 multiplications on matrices of size $n/2 \times n/2$.
- We can recursively break each of the 8 smaller matrix products into operations on matrices of size $n/4 \times n/4$.
- In general,we can recursively break the matrix product into smaller and smaller products, until we bottom out doing scalar multiplies.
- This general approach – break a big problem into one or more smaller subproblems of the same shape, solve them, and put the solutions back together – is what we call *divide and conquer*.
- The *hope* is that, since it is cheaper to solve each of the small problems than the full problem, divide and conquer might be faster than just brute-forcing the full problem.

So what does the D&C algorithm actually cost?

- We have a technique to compute that – recurrences!

- Suppose it costs $T(n)$ to multiply two $n \times n$ matrices by our recursive method.
- Adding two matrices of size $n/2 \times n/2$ takes time $\Theta(n^2)$.
- Hence, we have that the recursive approach takes time

$$T(n) \leq 8T(n/2) + cn^2$$

  for some constant c.
- Assume each base case takes constant time c_0.
- We can solve this recurrence to get a closed-form asymptotic expression for $T(n)$, using, e.g., the recursion tree method or the Master Method.
- For fun, let's do the recursion tree...

- Conclude that $T(n) = c_0 n^3 + \sum_{i=0}^{\log(n)-1} 2^i cn^2$, which works out to $\Theta(n^3)$.

Rats – subdividing didn't solve the problem asymptotically faster than the naive algorithm. Why not?

3 A Delicate Balancing Act

- When we try to break up a problem recursively, we do three kinds of work:
  1. dividing the problem into smaller subproblems, and combining the sub-solutions to get the final solution;
  2. recurring on smaller subproblems;
  3. work done for the base case whenever the recursion bottoms out.
- The first level of dividing and combining on the original problem instance is "top-of-tree" work – it happens once, at the root of the recursion tree.

- The base case work, which is usually constant-time each time it happens, is "bottom-of-tree" work – it happens at each of the bottom-most nodes of the recursion tree.
- Everything else is "middle-of-tree" work.
- In our matrix multiplication recursion, the top-of-tree work is $\Theta(n^2)$ (the four chunk-wise adds), but the bottom-of-tree work is $\Theta(n^3)$!
- The n^3 arises because we subdivide into 8 subproblems of size $n/2$, so there are a total of $8^{\log_2 n} = n^3$ base-case calls, each of which needs at least a few instructions to discover that it is a base case (if nothing else).
- Because of this bottom-of-tree work, *we can't possibly run asymptotically faster than the naive* $\Theta(n^3)$ *algorithm* so long as we use 8 subproblems of size $n/2$.
- **Principle**: to obtain a faster divide-and-conquer solution to a problem, the top-of-tree and bottom-of-tree work must both cost asymptotically less than the naive, non-recursive approach.
- In our case, the bottom-of-tree work killed us.
- The top-of-tree work is only $\Theta(n^2)$, so it is not a limiting factor.

NB: the Master Method for recurrences is a great quick-and-dirty way to check whether your strategy for subdividing a problem could improve on the naive algorithm. If the top- or bottom-of-tree work dominates, the MM's solution will be the cost of this work.

4 A Better Way: Strassen's Algorithm

- Can we repair our divide-and-conquer matrix multiply to run faster than $\Theta(n^3)$?
- Suppose I could magically compute the solution from using only *seven* recursive multiplies of size $n/2 \times n/2$, instead of the eight we used above.
- Now what does the algorithm cost?
- The only change to the recursion tree is that we have 7^i nodes at level i, rather than 8^i.
- Hence, the bottom-of-tree work is now $c_0 7^{\log_2 n} = n^{\log_2 7}$, or about $\Theta(n^{2.81})$.
- The top-of-tree work is still $\Theta(n^2)$.
- Have we improved the overall running time?

- Doing the accounting, the total work for the whole tree is now

$$\begin{aligned}
T(n) &= c_0 n^{\log_2 7} + \sum_{i=0}^{\log(n)-1} cn^2 \cdot (7/4)^i \\
&= c_0 n^{\log_2 7} + cn^2 \frac{(7/4)^{\log_2 n} - 1}{7/4 - 1} \\
&= c_0 n^{\log_2 7} + 4c/3n^2(n^{\log_2 7/4} - 1) \\
&= c_0 n^{\log_2 7} + 4c/3n^2(n^{\log_2(7)-2} - 1) \\
&= c' n^{\log_2(7)} - 4c/3n^2 \\
&= \Theta(n^{\log_2 7}).
\end{aligned}$$

- (We could have gotten this result a lot faster via the Master Method.)
- So yes, this change would make the algorithm asymptotically faster!

OK, bring on the magic!

- Volker Strassen (1969) came up with the following equivalences.
- Let

$$\begin{aligned}
P_1 &= A^{11}(B^{12} - B^{22}) \\
P_2 &= (A^{11} + A^{12})B^{22} \\
P_3 &= (A^{21} + A^{22})B^{11} \\
P_4 &= A^{22}(B^{21} - B^{11}) \\
P_5 &= (A^{11} + A^{22})(B^{11} + B^{22}) \\
P_6 &= (A^{12} - A^{22})(B^{21} + B^{22}) \\
P_7 &= (A^{11} - A^{21})(B^{11} + B^{12}).
\end{aligned}$$

- Then one can prove that

$$\begin{aligned}
C^{11} &= P_5 + P_4 - P_2 + P_6 \\
C^{12} &= P_1 + P_2 \\
C^{21} &= P_3 + P_4 \\
C^{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}$$

- (Feel free to plug in the definitions of the P's and check these four equivalences.)
- There are only *seven* multiplies on chunks of size $n/2 \times n/2$ required to compute P_1 through P_7.
- This is just as we analyzed above.
- Of course, there are 10 submatrix additions needed to set up these seven multiplies, and a further 8 additions needed to derive C from the P's.
- Still, each addition is $\Theta(n^2)$, so any constant number of adds doesn't change the asymptotic complexity of the algorithm!

- Hence, Strassen's algorithm is still $\Theta(n^{2.81})$ – better than brute force!

Algorithm design is a little bit structured and a little big magic. I'm teaching structural principles (i.e. divide-and-conquer, and what to shoot for in your recurrence). The magic specific to each problem is what makes algorithms hard/fun.

5 A Little More History

- Strassen's $n^{2.81}$ result (1969) was the first nontrivial improvement on $\Theta(n^3)$ matrix multiplication.
- Subsequently, Coppersmith and Winograd (1990) came up with a different approach that improved the cost to $\Theta(n^{2.375})$ (again, the exponent is approximate – the actual value is not a rational number).
- Subsequent work by Stothers (2010), Williams (2011), and Le Gall (2014) got this down to about $\Theta(n^{2.373})$.
- Matrix multiply by *any* algorithm costs $\Omega(n^2)$, since we need to look at all $\Theta(n^2)$ values in the input matrices to do it.
- But we don't know whether there exists an algorithm that can get us closer to n^2.

This is the state of the theory. What about practice?

- Nobody actually uses Coppersmith-Winograd or its improvements.
- The constant factors are too big to yield a practical improvement except for ridiculously huge matrices.
- Strassen's algorithm, if carefully implemented, can be a practical improvement on the naive algorithm if your matrices have n around a thousand.
- A good practical strategy for big matrices is to do a couple of Strassen iterations to get the subproblem size down a lot, then switch to the naive algorithm when the constant factors start to favor it.