Using Methodology to (hopefully) Design Clever Algorithms
CSE 202 Lecture 3

Lecture given by Russell Impagliazzo, scribed by Sashka Davis

September 29, 2005

Abstract

In this lecture we focus on using methodology to design clever algorithms. Our specific example will be a Divide and Conquer algorithm for large integer multiplication. Once we have the concrete algorithm next we abstract away the specific details. This will define a larger algorithmic search space, parameterized by the branching parameter b (in our case this is the number of partitions of the DnC large integer multiplication). In the parameterized algorithmic search space we find the optimal algorithm by deriving optimal value for the parameter b.

1 Outline

We continue with (large) integer multiplication. Last time we saw that if we break the problem into two parts we obtain an algorithm with running time $T(n) = 3T(n/2) + O(n)$. This is a bottom heavy recurrence with closed form $O(n^{\log_2 3})$. The question we ask: *if we break the problem into b pieces, each of size n/b, can we obtain asymptotically better algorithm?*

The two integers $x = \sum_{i=0}^{b-1} X_i 2^{\frac{n}{b}i}$, $y = \sum_{i=0}^{b-1} Y_i 2^{\frac{n}{b}i}$ are represented as degree $b-1$ polynomial as follows:

$$P_x(z) = \sum_{i=0}^{b-1} X_i z^i, \quad P_y(z) = \sum_{i=0}^{b-1} Y_i z^i$$

where the coefficients $|X_i| = |Y_i| = n/b$ bits, for all i. Note that the original inputs are recoverable from these polynomials by evaluation at $z = 2^{\frac{n}{b}}$.

We start with coefficient representation of the two polynomials, then will evaluate the polynomials at $2b-1$ points. Then we multiply the point-value representations of the polynomials point-wise. Here we have obtained a polynomial of degree $2b-2$ and to obtain the coefficient representation we need $2b-1$ points. Next we interpolate back to obtain the $2b-1$ coefficients. This gives rise to the following recurrence $T(n) = (2b-1)T(n/b) + O(n)$ and $T(n) \in O(n^{\log_b 2b} = O(n^{1+\frac{1}{\log_2 b}})$. We see that as b gets large we obtain asymptotically better time complexity. We want to find the *optimal* b for this problem using the DnC framework for large integer multiplication from last time.

Looking forward we will see that we cannot hope to get a linear time algorithm by this method. The best algorithm for integer multiplication is $O(n \log n \log\log n)$, and we will work towards something close to that in a few lectures.

2 The algorithm and time analysis

1. **Partition the problem**
   Given two n-bit integers x and y. Split the input into b pieces:

$$P_x(z) = \sum_{i=0}^{b-1} X_i z^i, \quad P_y(z) = \sum_{i=0}^{b-1} Y_i z^i$$

where the coefficients $|X_i| = |Y_i| = n/b$ bits, for all i.
This step takes time $O(n)$.

2. **Obtain a point value representation**
   Evaluate each poly at $2b-1$ points: $z \in \{-(b-1), \ldots, -1, 0, 1, \ldots, (b-1)\}$. Note that for the given range of values for z we need $O(\log b)$ bits to represent z. We do this using Horn's rule:

   INITIALIZE $T \leftarrow X_{b-1}$
   FOR $i = b-2$ DOWNTO 0 DO
   $\quad T \leftarrow T \times z$
   $\quad T \leftarrow T + X_i$
   OUTPUT T.

   z is a $O(\log b)$ bit integer and each X_i is n/b bit integer. The total cost of all additions is $O(b\frac{n}{b}) = O(n)$. The cost of each multiplication is $O(\log b \cdot \frac{n}{b})$, there are b iteration of the loop hence the total cost of all multiplications is $O(n \log b)$. Now remember that we do that for both $P_x(z)$ and $P_y(z)$ on $2b-1$ points, so the total cost of evaluation is $O((2b-1)n \log b) = O(bn \log b)$.

3. **Multiply the two polynomials point-wise recursively**
   $P_{xy}(z) = P_x(z) \times P_y(z)$ at $z \in \{-(b-1), \ldots, -1, 0, 1, \ldots (b-1)\}$.
   Here we have $(2b-1)$ calls with arguments of size n/b, hence the total cost is $(2b-1)T(\frac{n}{b})$.

4. **Interpolate step**
   We use Lagrange interpolation formula to obtain from $2b-1$ points a polynomial of total degree $2b-2$ but we will elaborate this step a bit, so that we derive precisely the time complexity.
   Define $(2b-1)$ polynomials $q_i(z)$ such that $g_i(i) = 1$ and 0 everywhere else $(g_i(j) = 0, i \neq j)$, for $i \in \{-(b-1), \ldots, -1, 0, 1, (b-1)\}$.

   $$q_i(z) = \frac{\Pi_{k \neq i}(z-k)}{\Pi_{k \neq i}(i-k)}$$

   The previous step has calculated $P_{xy}(i), i \in \{-b-1), \ldots -1, 0, 1, \ldots,)b-1)$
   Now we have the coefficients representation of P_{xy} as

   $$P_{xy}(z) = \sum_i q_i(z) P_{xy}(i)$$

   Analysis: $P_{zy}(z)$ is a linear combination of $2b-1$ rationals. Each coefficient q_i is of size $O(b \log b)$ (because we have $2b-1$ multiplications of $O(\log b)$ bit numbers) and takes $O(b^2 \log^2 b)$ time to compute (using repeated addition).
   Total time during interpolation is $\sum(\frac{n}{b} \cdot b^2 \log b) = O(b^2 n \log b)$.

3 Time complexity analysis

The recurrence expressing the time complexity of the algorithm above is:

$$T(n) \leq 2bT(n/b) + O(b^2 n \log b)$$

First note that we can never obtain a linear time algorithm using this approach, since the additive term is super-linear whenever b is more than a constant. However we can still optimize to find the best value for b by obtaining a closed form for the recurrence and finding the best value for b. We unwind the recurrence above, using the same parameter b inside all the recursive calls.

Top level: we have cost $O(nb^2 \log b)$

Level 2: we have 2b calls each of cost $O(\frac{n}{b} b^2 \log b)$ hence the total cost at level 2 is $O(2nb^2 \log b)$

$\vdots$

Level i: we have $2^i b^i$ calls each of cost $O(\frac{n}{b^i} b^2 \log b)$ hence the total cost at level i is $O(2^i nb^2 \log b)$

The recurrence is bottom heavy, at each level we have twice the amount of work done at the previous level, hence the total cost is the work done at the last level, namely

$$O(n2^{\log_b n} b^2 \log b) = O(n2^{\log_b n + 2\log_2 b + \log\log_2 b})$$

The terms in this cost are balanced when $b = 2^{\sqrt{\log n}}$, giving us a total time of $O(n2^{3\sqrt{\log n}}\sqrt{\log n})$.

4 Conclusions

To rehash what we have done so far is:

1. First we find an algorithm for the problem.
2. We abstract away the specifics and try to generalize (or call it parameterize) out algorithm.
3. This process of abstraction generates a class of algorithms and we want to find the best algorithm within this algorithmic search space.

A further step, would be to parameterize further, perhaps introduce another parameter and hence broaden the algorithmic search space.

5 Off the track..

Is true that for any computable function f there exists an optimal algorithm computing it? The answer is negative. There exists a recursive function for which no optimal algorithm exists [1]. More precisely:

Theorem 1 *(Blum Speedup Theorem) There exists a total recursive function f with the property that to every Z_i algorithm that computes f in time $O(\Phi_i(n))$ steps there exists an algorithm Z_j that computes f in time $O(\log \Phi_i(n))$ steps.*

The above theorem states that there is an infinite sequence of algorithms for this special function f, each exponentially faster than the previous.

Just a thought on this result: Note that f must be *very hard* to compute. Consider the kinds of functions we compute in this class. Usually they have algorithms that even if we use brute-force approach it would take, in the worst case, $O(poly(n) \cdot 2^n)$ (if we are examining all possible subsets of the input, say clique, or independent set problems), or $O(poly(n) \cdot n!)$ (if we are examining all possible permutations of the input, say traveling salesman problem). Obviously, any of these functions are not candidates for f in the above theorem (after we log once n^n we get $n \log n$, and applying the log twice gives a sublinear time function).

References

[1] Manuel Blum, *A Machine-Independent Theory of Complexity of recursive Functions*, Journal of ACM, 14(2), 1967.