CSOR W4231.002 **Lecture 2: February 3, 2014** **Spring 2014**

Today we will discuss the divide and conquer paradigm. The paradigm is applied in three steps:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- Solve the subproblems separately by using recursion. When the size of the subproblems is small enough, "bottom out" the recursion by solving the subproblems in a straightforward manner.
- **Combine** the solutions to the subproblems into an overall solution.

The analysis of the running time of a divide and conquer algorithm generally involves solving a **recurrence relation** that bounds the time spent solving the original instance recursively in terms of the time spent on solving the smaller instances and the time spent in combining the solutions.

1 Application: Merge-Sort

Applying the divide and conquer principle to the problem of sorting we obtain the following algorithm.

- Divide the input list into two lists of equal size.
- Solve the two sorting subproblems on these lists separately by using recursion. When the size of the subproblems is reduced to 1 do nothing (an array of size 1 is trivially sorted). [1]
- Combine the two results into an overall solution.

This algorithm is formalized below in procedure Merge-Sort. An array A of size n is used to store the input list. Merge-Sort takes as input the array A, as well as the region of A that will be sorted once the procedure is executed; this region is defined by the limits $left$ and $right$. Hence the output of Merge-Sort is the sorted subarray $A[left, \dots, right]$. The initial call is Merge-Sort$(A, 1, n)$.

```
Merge-Sort ($A, left, right$)
  if $size(A) = 1$ then return //$A$ is an array with one element, thus trivially sorted
  end if
  $middle = left + \lfloor (right - left)/2 \rfloor$
  Merge-Sort ($A, left, middle$)
  Merge-Sort ($A, middle + 1, right$)
  Merge ($A, left, middle, right$)
```

Subroutine Merge involves merging two *sorted* lists (arrays) of sizes $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ respectively, into one sorted list of size n. How can we accomplish that?

1.1 Subroutine Merge

Intuition: To merge two sorted lists (arrays) of sizes $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$, repeatedly

- compare the two items in the front of the two lists
- copy the smaller item to the output; update the "front" of the lists; if the end of a list is reached, copy the rest of the other list to the output.

[1]**Note:** It is more efficient to "bottom out" the recursion when the lists have size 2, in which case we sort the two elements by simply comparing them.

This intuition is formalized in the algorithm below. The following notation is used.
1. $|X|$ denotes the number of elements in array X.
2. The statement $A[i, \ldots, j] = B[i, \ldots, j]$ is a shorthand for the loop **for** $k = i$ to j **do** $A[k] = B[k]$; hence this statement requires $O(j - i)$ time.

```
Merge($A, left, mid, right$)
  $L[1, \ldots, mid - left + 1] = A[left, \ldots, mid]$
  $R[1, \ldots, right - mid] = A[mid + 1, \ldots, right]$
  $pointer1 = 1$
  $pointer2 = 1$
  while $pointer1 <= |L|$ and $pointer2 <= |R|$ do
      if $L[pointer1] <= R[pointer2]$ then
          $A[index] = L[pointer1]$
          $pointer1 = pointer1 + 1$
      else
          $A[index] = R[pointer2]$
          $pointer2 = pointer2 + 1$
      end if
  end while
  if $(pointer1 > |L|)$ then $A[index, \ldots, right] = R[pointer2, \ldots, right - left]$
  else if $(pointer2 > |R|)$  $A[index, \ldots, right] = L[pointer1, \ldots, mid - left + 1]$
  end if
```

Note that Merge requires extra $\Theta(n)$ space for the temporary arrays L, R where copies of the two lists that will be merged are stored. The output of Merge is stored directly in A.

1.2 Analysis of Merge

1. **Correctness:** Follows from the fact that the smaller number in the input is $L[1]$ or $R[1]$ and it will be the first number in the output. The rest of the output is just the list obtained by Merge(L, R) *after* deleting the smallest element.

2. **Running time:** At the top call of Merge-Sort$(A, 1, n)$, lists L, R have $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ elements respectively. Note that one element of either list may be used in many comparisons (e.g., if every element of L is smaller than every element of R). However, in each execution of the while loop, a comparison is performed, and one element from L or R is appended to the output. This element will never be considered again in the future. All elements from both lists will have been appended to the output after at most $\lfloor n/2 \rfloor + \lceil n/2 \rceil - 1 = n - 1$ iterations. Since the work within each iteration takes constant time (compare two elements, store an element to the output, increment an index), the while loop of Merge together with the two conditional statements below require $O(n)$ time. Initializing lists L, R (the first two lines of the code) requires $O(n)$ time as well. Hence Merge takes time cn for some constant $c > 1$ to merge two lists of size $n/2$.

1.3 Analysis of Merge-Sort

1. **Correctness:** For simplicity assume that $n = 2^k$, for integer $k \geq 0$.

   - **Basis:** For $k = 0$ there is one element in A (hence A is sorted) and Merge-Sort does nothing.
   - **Induction Hypothesis:** Assume that Merge-Sort correctly sorts any array of size 2^k for some $k > 0$.
   - **Induction Step:** We will show that Merge-Sort correctly sorts an array of size 2^{k+1}. Merge-Sort will recursively invoke itself twice; first, on input the $2^{k+1}/2 = 2^k$ elements in the left

half of A and then on input the $2^{k+1}/2 = 2^k$ elements in the right half of A. By the induction hypothesis, these two recursive calls of Merge-Sort will return both halves of A correctly sorted. At the last step of Merge-Sort, it follows by the correctness of Merge that Merge will correctly merge the two sorted arrays of size 2^k into one sorted output array of size $2 \cdot 2^k = 2^{k+1}$. Hence the output of Merge-Sort is the sorted array of size 2^{k+1}.

2. Running time of Merge-Sort: The running time of Merge-Sort satisfies the following **recurrence relation**:

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn, n \geq 2 \qquad (1) \\ T(1) &\leq c \end{aligned}$$

2 Recurrences

Equation (1) is a typical recurrence relation: an inequality (or equation) bounds $T(n)$ in terms of an expression involving $T(m)$ for $m < n$. In general we can ignore floor and ceiling notations: asymptotic bounds are not affected by them (only exact solutions). Equation (1) does not provide an asymptotic bound on $T(n)$; to obtain an explicit bound, we need to solve the recurrence relation so that $T(n)$ appears only on the left-hand side.

Henceforth, we will be using log for $\log_2$ and ln for $\log_e$. If we want to use a different basis $b \neq 2$ for our logarithms, we will denote this explicitly by $\log_b$.

2.1 Recursion Trees

This approach consists of analyzing the tree of recursive calls. It involves 3 steps: (1) analyze the first few levels of the tree, (2) identify a pattern and (3) obtain a bound on the running time by summing over all levels of the tree the time spent on each level.

For example, analyzing the recursion tree for equation (1) shows that at level i there are 2^i subproblems, each requiring time $(cn/2^i)$ for $i \geq 0$. The last level of the tree is level $\log_2 n$ where each subproblem has size 1 and takes time $T(1)$. Hence we conclude that

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i \cdot \frac{cn}{2^i} = cn \cdot (\log_2 n + 1) = O(n \log n).$$

2.2 The substitution method

Sometimes we may have a good initial guess for the asymptotic behavior of a recurrence. For example, we may have encountered and solved a similar recurrence before. In this case, we may *guess* a bound and then use induction to prove that our guess is correct.

As an example consider the recurrence given by equation (1). We believe that $T(n) = O(n \log n)$ so we guess that $T(n) = cn \log n$ for all $n \geq 1$.

To verify our guess we will use strong induction on n. Strong induction is another form of induction where the induction step at n requires that the inductive hypothesis holds at all steps $1, 2, \ldots, n-1$ and not just at step $n-1$, as with simple induction.

Base case: $n = 1$: $T(1) \leq c \log 1 = 0$; we know that $T(2) \leq c$, hence the induction basis does not hold.

This does not mean that our guess is wrong: since we are looking for an asymptotic bound on $T(n)$, we only want our bound to hold for all $n \geq n_0$ for some constant n_0 of our choice. In general, choosing a larger n_0 and a larger constant factor in front of the high order term ($n \log n$ in this case) will make our induction work if our original guess was correct.

In our example, pick $n_0 = 2$. For $n = 2$, equation (1) yields $T(2) \leq 2c + 2c = 4c$. Let $k = 4c$; then $T(2) \leq k$. We will show by strong induction that the recurrence given by equation (1) satisfies $T(n) \leq kn \log n$ for all $n \geq 2$. Note that $kn \log n$ has the same asymptotic behavior as $cn \log n$.
Base case: $n = 2$: $T(2) \leq 2k \log 2 = 2k$ and we know that $T(2) \leq k$; hence the induction basis is true.
Induction Hypothesis: For all $2 \leq m < n$, assume that $T(m) \leq km \log m$.
Induction Step: We will show that $T(n) \leq kn \log n$. By induction hypothesis, for $m = n/2$, $T(n/2) \leq (kn/2) \log (n/2)$. Then

$$T(n) \leq 2T(n/2) + cn \leq 2(kn/2)\log(n/2) + cn \leq 2(kn/2)\log(n/2) + 4cn = kn\log n - kn + kn = kn \log n$$

2.3 The Master Theorem

Theorem 1 *Let $a \geq 1$, $b \geq 2$ be integers and $c, k > 0$ be constants. Let $T(n)$ be defined over the non-negative integers by the recurrence*

$$T(n) = aT(n/b) + cn^k, \tag{2}$$

where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ is asymptotically bounded as follows:

1. *$T(n) = O(n^{\log_b a})$ if $a > b^k$.*
2. *$T(n) = O(n^k \log n)$ if $a = b^k$.*
3. *$T(n) = O(n^k)$ if $a < b^k$.*

Sketch of the proof: Construct the recursion tree for the recurrence of equation 2. There are $\log_b n$ levels; at level i, there are a^i subproblems, each of which requires $\left(\frac{n}{b^i}\right)^k$ time. The total time $T(n)$ spent at all levels of recursion is given by

$$\sum_{i=0}^{\log_b n} a^i \cdot \left(\frac{n}{b^i}\right)^k = n^k \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i.$$

The master theorem follows by computing this sum for the three possible outcomes of the comparison between a and b^k.

3 Binary Search: Example of sublinear time

Input: sorted list A of n integers, integer x
Output: index j s.t. $1 \leq j \leq n$ and $A[j] = x$; or "no" if x is not in A.

Idea: use the fact that the array is sorted and probe specific entries.
Let $mid = \lceil n/2 \rceil$. If $A[mid] = x$ then return mid; else if $A[mid] > x$ then throw away the right half of A and $A[mid]$, and continue recursively with $A[1, \ldots, mid - 1]$; else (that is, $A[mid] < x$), throw away the left half of A and $A[mid]$, and continue recursively with $A[mid + 1, n]$.

3.1 Analysis of running time

Note that at each step there is a region of A where x could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If n is odd, we throw away $\lceil n/2 \rceil$ elements.
- If n is even, we throw away at least $n/2$ and at most $n/2 + 1$ elements.

If we consider a comparison as an elementary computational step, then the recurrence for the running time of binary search is

$$T(n) \leq T(n/2) + 2.$$

By the Master Theorem, for $b = 2, a = 1, k = 0$, we obtain $T(n) = \Theta(\log n)$.
Note that in order to obtain running time $O(\log n)$, we made use of the following facts:
1. Arrays allow for random access: given an index, the corresponding array entry is read in $O(1)$ time.
2. At each iteration our algorithm does a *constant amount of work* (here we use the previous fact) to throw away a *constant fraction* of the input.

4 Integer Multiplication

In primary school, we learned the following algorithm for multiplying two integers x and y:
1. Compute n partial products by multiplying every digit of y separately with x, starting from the lowest-order digit.
2. Add up all the partial products.
This method works the same for decimal or binary numbers. As an exercise, work it out for $(12)_{10} \cdot (11)_{10}$ and $(1100)_2 \cdot (1011)_2$. In what follows we will consider binary x, y.

In this setting, it is reasonable to count a single operation on a pair of bits as a primitive computational step. Then the time to compute one partial product is $O(n)$. There are n partial products hence the algorithm spends $O(n^2)$ time to compute all of them. Another $O(n^2)$ time is used to add up all the partial products together to obtain the final result.

Can we do better than $O(n^2)$ or is $\Omega(n^2)$ a fundamental lower bound on the complexity of multiplying two integers?

4.1 Karatsuba multiplication

In 1960, Russian mathematician Karatsuba, a 23-year-old student then at the Moscow State University, came up with the following divide and conquer approach.

Rewrite each number as the sum of the $n/2$ high-order bits and the $n/2$ low-order bits. Hence

$$x = x_1 2^{n/2} + x_0,$$
$$y = y_1 2^{n/2} + y_0,$$

where $x_1, x_0, y_1, y_0 \leq 2^{n/2}$, i.e., they are $n/2$-bit numbers.

- E.g.: $x = (1100)_2 = (11)_2 \cdot 2^2 + (00)_2 = x_1 \cdot 2^{n/2} + x_0$, for $n = 4, x_1 = (11)_2, x_0 = 0$
- E.g.: $y = (1011)_2 = (10)_2 \cdot 2^2 + (11)_2 = y_1 \cdot 2^{n/2} + y_0$, for $n = 4, y_1 = (10)_2, y_0 = (11)_2$

Then

$$x \cdot y = (x_1 2^{n/2} + x_0) \cdot (y_1 2^{n/2} + y_0) = \overbrace{\underbrace{x_1 \cdot y_1}_{P1} \cdot 2^n}^{T1} + \overbrace{(\underbrace{x_1 \cdot y_0}_{P_2} + \underbrace{x_0 \cdot y_1}_{P_3}) \cdot 2^{n/2}}^{T2} + \underbrace{x_0 \cdot y_0}_{P_4} \tag{3}$$

Thus we reduced the problem of 1 instance of size n (i.e., computing the product $x \cdot y$ where x, y are n-bit numbers) to the problem of solving 4 instances of size $n/2$ (i.e., computing the four products $P_1 = x_1y_1, P_2 = x_1y_0, P_3 = x_0y_1$ and $P_4 = x_0y_0$, where x_1, x_0, y_1, y_0 are $n/2$-bit numbers).

This is a divide and conquer solution:

- recursively solve the 4 subproblems P_1, P_2, P_3 and P_4;

- combine the solutions to the 4 subproblems as follows:
  1. T_1 is computed by multiplying P_1 by 2^n; this is done by shifting x_1y_1 to the left by n positions; thus computing T_1 once P_1 has been computed requires $O(n)$ time;
  2. T_2 is computed by first adding P_2 and P_3: this takes $O(n)$ time (addition between two $O(n)$-bit numbers); then multiplication by $2^{n/2}$ is performed by shifting $(x_1 \cdot y_0 + x_0 \cdot y_1)$ to the left by $n/2$ positions which also requires $O(n)$ time; hence T_2 requires $O(n)$ time;
  3. Adding the three terms T_1, T_2, P_4 together takes time $O(n)$ (addition of three numbers with at most $2n$ bits).

Then the running time of this algorithm is given by the following recurrence

$$T(n) \leq 4T(n/2) + cn,$$

for constant $c > 1$. By the Master Theorem $T(n) = O(n^2)$. Hence we did not achieve any improvement over the primary school algorithm.

However if we could use only 3 $n/2$-bit multiplications we would obtain the recurrence

$$T(n) \leq 3T(n/2) + cn,$$

which, by the Master Theorem, yields $T(n) = O(n^{1.59})$ and thus asymptotically improves upon the primary school multiplication method!

The key observation to achieve this is that we do not really need all of P_1, P_2, P_3 and P_4. What we really need is P_1, $P_2 + P_3$ and P_4. There is a clever way to compute these three terms by only using 3 multiplications between $n/2$-bit numbers; compute

- $P_1 = x_1y_1$,
- $P_4 = x_0y_0$,
- $P_5 = (x_1 + x_0)(y_1 + y_0)$

and use the fact that

$$P_5 = (x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_0y_0 + x_1y_0 + x_0y_1 = P_1 + P_4 + (P_2 + P_3)$$

to deduce that

$$P_2 + P_3 = P_5 - P_4 - P_1.$$

Hence we rewrite equation 3 as

$$x \cdot y = (x_1 2^{n/2} + x_0) \cdot (y_1 2^{n/2} + y_0) = P_1 \cdot 2^n + (P_5 - P_4 - P_1) \cdot 2^{n/2} + \ P_4,$$

where only three $n/2$-bit integer multiplications are performed (namely, P_1, P_4, P_5) and the time for combining the solutions to the subproblems remains $O(n)$ since we have two more subtractions and one more addition between n-bit numbers than we had in equation 3. Hence

$$T(n) \leq 3T(n/2) + cn = O(n^{1.59}).$$

Note that, when we recursively compute $(x_0 + x_1)(y_0 + y_1)$, each of $x_1 + y_0$, $x_0 + y_1$ might be an $(n/2+1)$-bit number. This does not affect our asymptotic analysis. Also, when n is small enough, there is no reason to continue with recursion: the conventional algorithm is probably more efficient since it uses fewer additions.