Lecture 11: Greedy algorithm and divide and conquer

In this class, we first finish the last example of greedy algorithm: scheduling with deadline. I demonstrate that the simple greedy strategy, prefer tasks with earlier deadlines, actually is optimal. Our textbook covers this well in section 4.2 (page 125). Check out this section if you did not get the full idea.

We now study how to analyze divide and conquer algorithms. For the merge sort, its running time can be written as $T(n) = 2T(n/2) + \Theta(n)$, when $n > 1$. When $n = 1$, $T(n) = \Theta(1)$. There are several methods to estimate $T(n)$. The first is called direct substitution. This method will repetitively get rid of the sub-problems by substituting with even smaller sub-problems. Here, $T(n) = 2T(n/2) + cn$, $T(n/2) = 2T(n/4) + cn/2$, $T(n/4) = 2T(n/8) + cn/4$, and so on. Then, we get rid of $T(n/2)$ term by substituting it with $2T(n/4) + cn/2$, and so on. This will lead to $T(n) = nT(1) + cn\log n$, because we stop after $\log n$ substitutions and we get sub-problems of size 1. If you are unclear about this step, you should write down these recurrences and try it yourself.

The second approach is to first guess a solution and then prove your guess by induction. Let us consider the merge sort algorithm again. We make a guess: $T(n) \leq cn\log_2 n$. For $n = 2$, $T(2) \leq 2c$, if we pick $c$ to be large enough. Now assume $T(m) \leq cn\log m$ for all $m < n$. This means $T(n/2) \leq cn/2\log_2(n/2)$. Then, we have: $T(n) \leq 2T(n/2) + cn \leq cn\log_2(n/2) + cn = cn\log_2 n – cn + cn = cn\log_2 n$.

The concept of so-called recursion tree can be useful in visualizing divide and conquer algorithms. The recursion tree is divided into levels. Nodes of a level are labeled with the work spent on this level (i.e. the divide and combine work). The conquer portion of the work is expressed its descendant nodes. For the merge sort, each node for problem size of $n/2^k$ has work $cn/2^k$. There are $\Theta(\log n)$ levels in the tree, because each time we go down the tree by one level, the number of subproblems doubles. At the bottom, there are $n$ subproblems (because in merge sort, the subproblems are disjoint).

Lecture 12: Divide and conquer: solving recurrence and design of algorithms

A third approach of analyzing recurrences is to use the Master theorem. Unfortunately, our textbook does not cover the Master theorem in an easy-to-use way. I would suggest you to read carefully the document on Master theorem posted on the class webpage. Note: you may need Master theorem in homework and exams. Two things to remember: remember the small $\epsilon$ in cases 1 and 3, and also the regularity condition for case 3. Intuitively, the Master theorem says you should compare $f(n)$ (the divide and combine time) with $n^{\log_b a}$ (where $b$ specify how much the subproblems are shrunk and $a$ is the number of subproblems). Essentially, case 1 says if $f(n)$ is dominated by $n^{\log_b a}$ (by a polynomial gap of $\epsilon$), then the total time is $\Theta(n^{\log_b a})$. Case 2 says if the two are more or less the same, then we should multiple by a $\log(n)$ factor to one of them to get the overall time. Case 3 says if $f(n)$ dominates $n^{\log_b a}$ (by a polynomial gap of $\epsilon$), then the total time is simply $\Theta(f(n))$ (if $f(n)$ passes the regularity condition, which most functions we will use do). You should get yourself familiar with the Master theorem by trying some examples. In class, we did some examples. Make sure you know how to solve them.

Divide and conquer is a widely used algorithm design paradigm. We now continue to show how to design a divide and conquer algorithm.

Our first example is the MaxProfit problem, where you are given $n$ days’ stock price $P[1..n]$ and you want to make the most profit by buying a fixed number of shares at day $b$ and later selling at day $s$.

A naive algorithm would try all $\binom{n}{2}$ pairs of days, which leads to an $O(n^2)$ algorithm. We now use divide and conquer. We define $MaxProfit(l,r)$ as the maximum profit we can earn by buying
and selling within the period of \([l, r]\). Note we must first buy then sell. Then, the solution is simply 
\(\text{MaxProfit}(1, n)\). Here is the algorithm.

1: MaxProfit(l, r)
2: if \(l = r\) then
3:    return 0.
4: end if
5: \(m = \lfloor (l + r) / 2 \rfloor\).
6: \(m_1 \leftarrow \text{MaxProfit}(l, m)\)
7: \(m_2 \leftarrow \text{MaxProfit}(m + 1, r)\)
8: \(v_1 \leftarrow \text{MIN}(P[l], P[l + 1], \ldots, P[m])\)
9: \(v_2 \leftarrow \text{MAX}(P[m + 1], P[m + 2], \ldots, P[r])\)
10: return \(\text{MAX}(m_1, m_2, v_2 - v_1)\).

This algorithm is correct because lines 6 and 7 cover the cases where you buy and sell within the first (resp. second) half of the problem, while line 8 covers the case when you buy before day \(m\) and sell after day \(m\). The running time \(T(n) = 2T(n/2) + \Theta(n)\), which leads to \(O(n \log n)\) time. Note, the combine step takes \(\Theta(n)\) to find \(v_1\) and \(v_2\).

Our next problem is on multiplying two \(n\) by \(n\) square matrices \(A\) and \(B\). We apply divide and conquer. See Section 4.2 for more details. One comment about analyzing the first divide and conquer algorithm. The combine step takes \(\Theta(n^2)\). This is because it involves adding up a constant number of \(n/2\) by \(n/2\) matrices (which takes \(\Theta(n^2)\)) time, and then copy these four resulting \(n/2\) by \(n/2\) matrices \(C_{i,j}\) into the \(n\) by \(n\) matrix \(C\), which also take \(\Theta(n^2)\) since there are \(\Theta(n^2)\) cells in \(C\) and each copying takes \(O(1)\) time. Thus, \(T(n) = 8T(n/2) + \Theta(n^2)\). We use the Master theorem to find out \(T(n) = \Theta(n^3)\), which is no better than the original naive algorithm. Surprisingly, Strassen found a way to improve the algorithm. See page 79 for more details on how to reduce the number of subproblems, and how to get better running time.

We now consider the integer multiplication problem: given two (binary) integers \(x\) and \(y\), compute the product of \(x\) and \(y\). Here is a primary school algorithm for this problem.

1: \(z \leftarrow 0\)
2: for \(i = 1\) to \(n\) do
3:    \(t \leftarrow x\) if \(y[i] = 1\), and 0 otherwise.
4:    Left-shift \(t\) by \(i - 1\) bit.
5:    \(z \leftarrow z + t\).
6: end for

Now we analyze this algorithm. We have \(n\) iterations of the loop. In each iteration, we are performing no more than \(2n\) single bit summations for adding two integers. Note: since we left shift \(t\), when properly implemented, we will still only add no longer than \(n\) bits in the addition. Thus, the algorithm will use \(n \times 2n = 2n^2 = O(n^2)\) single bit additions.

We can apply divide and conquer to design a new algorithm. We divide \(x\) (and \(y\)) into half: the high order \(n/2\) bits and the low order \(n/2\) bits. That is, we write \(x = x_1 \cdot 2^{n/2} + x_0\) and \(y = y_1 \cdot 2^{n/2} + y_0\). Then, we recursively solve four sub-problems (each of half size as before) and then combine into the product of \(x\) and \(y\). Note combine step takes \(O(n)\) since it involves adding a few \(O(n)\) bits integers, which can be done by \(O(n)\) time. Unfortunately, the Master theorem states that the running time is \(\Theta(n^2)\). However, there is a clever trick to speedup the computation.