Designing Algorithms by Induction

Let’s look at a run-time analysis for the Huffman algorithm from last time. Roughly it looks like:

Initialization
- For each $f_i$ create a 1-node tree labeled by $f_i$: $O(n)$ time
- Create a priority queue $Q$ of these trees based on frequencies: $O(n)$ time

The algorithm: $Huffman(Q, n)$

Pre-work
- Remove two lowest-frequency trees from $Q$: $O(\log n)$
- Merge them into a new tree and add this tree back to $Q$: $O(\log n)$

Work
- Call $Huffman(Q, n - 1)$ on smaller priority queue

Post-work
None.

What’s the run-time of the algorithm? Let’s pull out the one-time initialization step. Denote the worst-case runtime of $Huffman(Q, n)$ by $T(n)$. Then $T(n)$ equals Pre-work time plus time for recursive call:

$$T(n) \leq c \log n + T(n - 1)$$

Why not equality? We’re replacing the actual time needed to solve a particular sub-problem of size $n - 1$ with the worst-case run time.

This inequality is called a recurrence. They are a main part of our focus today. The other part is....

Divide & Conquer

Not all problems seem to admit greedy solutions. But a second basic design paradigm often works quite well.

Observation 1. Often a problem $P$ of size $n$ can be solved by dividing it into several small problems of the same type and appropriately combining their solutions to solve the original problem.

These algorithms all have similar form

Pre-work (Divide): Transform problem $P$ of size $n$ into problems $P_1, \ldots, P_k$ of sizes $n_1, \ldots, n_k$ (each $n_i < n$)

Work: Solve each of $P_1, \ldots, P_k$

Post-work (Conquer): Appropriately combine solutions of $P_1, \ldots, P_k$ to construct solution of $P$

The nature of the problem (or of the algorithm being proposed to solve it) often will suggest
- How many subproblems are required
- What their sizes are
- Whether the subproblems overlap
And these choices will impact the performance of the algorithm so developed.

Our goal is to illustrate the design paradigm through several examples, and learn the tools—most notably, recurrence relations—needed to analyze the performance of our algorithms.

**Sorting**

Most sorting algorithms are some form of Divide & Conquer

**Selection Sort:** Pre-work: Find largest element $x$; Work: SelectionSort rest of data; Post-work: Add $x$ to end of sorted rest

**Insertion Sort:** Pre-work: Select an element $x$; Work: InsertionSort rest of data; Post-work: Insert $x$ into sorted rest

**Merge Sort:** Pre-work: Divide data in half; Work: MergeSort each half; Post-work: Merge sorted halves.

**Quicksort:** Pre-work: Pick a pivot $x$ and partition data based on $x$; Work: Quicksort each partition; Post-work: Reassemble partitions (and $x$)

These algorithms perform differently for two main reasons: the time required for the pre- and post-work and the number/size of subproblems.

We can see this if we attempt to count operations: Let $T(n)$ represent the number of operations performed in the worst case for, say, Selection Sort. Then,

$$T(n) = \text{time for pre-work} + \text{time for work} + \text{time for post-work}$$

$$\leq c_1 n + T(n-1) + c_2$$

$$\leq cn + T(n-1) \text{ (assuming } n \geq 1 \text{ and } c = c_1 + c_2)$$

The inequality $T(n) \leq cn + T(n-1)$ is called a recurrence. We would like to use it to say something about $O(T(n))$.

Why is it an inequality?

- Can’t count # of operations exactly.
- Worst-case of size $n$ needn’t lead to worst case of size $n-1$!

Applying similar logic for Merge Sort gives;

$$T(n) = \text{time for pre-work} + \text{time for work} + \text{time for post-work}$$

$$\leq c_1 + T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + c_2 n$$

$$\leq c_1 + 2T(\frac{n}{2}) + c_2 n \text{ (assume for now that } n \text{ is a power of 2)}$$

$$\leq cn + 2T(\frac{n}{2}) \text{ (assuming } n \geq 1 \text{ and } c = c_1 + c_2)$$

Note: Recurrences need a base case (sometimes several). Here it’s just $T_1 \leq d$.

This gives the recurrence $T(n) \leq cn + 2T(\frac{n}{2})$. Which recurrence seems more promising?

Some observations
• We often cannot characterize $T(n)$ completely, but we can get an upper bound on its growth.
• We will 'cheat' by assuming (frequently) that our $n$ has a nice form (say $n = 2^k$) to simplify the recurrence.
• Thus $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$ becomes $2T(\frac{n}{2})$
• Is this reasonable? Well, assuming that $T(n)$ is an increasing function of $n$, then replacing $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$ with $2T(\frac{n}{2})$ would be fine, and it can be shown that removing the rounding up will not make an asymptotic difference.

Let’s try another example:

**Binary Search**

Suppose we have a sorted array $A[1..n]$ and we wish to know the location of some element $x$ in the list.

Pre-work Compare $x$ to middle element of $A$; if equal, halt : $c$ steps

Work If $x$ is less than middle element, binary search on $A[1..n/2 - 1]$, else binary search on $A[n/2 + 1..n]$:

$\leq T(n/2)$

Post-work None.
So $T(n) \leq c + T(\frac{n}{2})$

**Multiplication of $n$-bit Integers**

Multiplication of two $n$-bit integers seems to take $\Theta(n^2)$ time. Does it really? Can we do better?

Assume $n$ (the number of bits!) is a power of 2.

If $A$ is an $n$-bit integer then $A = x2^{n/2} + y$ where $x$ and $y$ are the integers consisting of the highest and lowest $n/2$ bits of $A$ respectively.

So, suppose now that we have two such numbers $A_1 = x_12^{n/2} + y_1$ and $A_2 = x_22^{n/2} + y_2$.

Then $A_1A_2 = (x_12^{n/2} + y_1)(x_22^{n/2} + y_2) = x_1x_22^n + (x_1y_2 + x_2y_1)2^{n/2} + y_1y_2$.

Note that the multiplication by powers of 2 just shifts bits to the right so it is easy to build $A_1A_2$ from these pieces.

Now let’s count operations: Let $T(n)$ be the number of operations required on two $n$-bit integers.

Pre-work Decompose $A_1$ and $A_2$ into $\{x_1, y_1\}$ and $\{x_2, y_2\}$: $cn$

Work Compute 4 products $x_1x_2, x_1y_2, x_2y_1, y_1y_2$: $\leq 4T(n/2)$

Post-work Perform shifts and merges: $dn$

This gives $T(n) \leq an + 4T(\frac{n}{2})$, where $a = c + d$.

**Solving Recurrences**

All of the recurrences presented have one of the two forms

• $T(n) \leq f(n) + T(n - a)$, where $a \in \mathbb{N}$, or
• $T(n) \leq f(n) + qT(\frac{n}{b})$, where $q, b \in \mathbb{N}$. [In fact, $b = 2$ in all of our examples so far.]
Thinking about Recurrence Relations: The Work Tree

The selection (and insertion) sort recursions both looked like $T(n) \leq cn + T(n-1)$, with $T() = d$. Let’s try to solve this recurrence.

$$T(n) = \leq cn + T(n-1)$$
$$\leq cn + c(n-1)T(n-2)$$
$$= c[n + (n - 1)] + T(n - 2)$$
$$\leq c[n + (n - 1) + \ldots + (n - (k - 1))] + T(n - k)$$
$$\ldots$$
$$\leq Cn^2$$

Consider the recurrence $T(n) \leq cn + 2T(\frac{n}{2})$. Let’s try evaluating it by unwinding it.

$$T(n) \leq cn + 2T(\frac{n}{2})$$
$$\leq cn + 2[c(\frac{n}{2}) + 2T(\frac{n}{4})] = c(n) + 4T(\frac{n}{4}) = c2n + 4T(\frac{n}{4})$$
$$\leq c2n + 4[c(\frac{n}{4}) + 2T(\frac{n}{8})] = c3n + 2^3T(\frac{n}{2^3})$$
$$\ldots$$
$$\leq ckn + 2^kT(\frac{n}{2^k})$$
$$\ldots$$
$$\leq c(log n)n + 2^{log n}T(1) = cn \log n + dn \in O(n \log n)$$

What if we try this with the integer multiplication problem?

$$T(n) \leq cn + 4T(\frac{n}{2})$$
$$\leq cn + 4[c(\frac{n}{2}) + 4T(\frac{n}{4})] = c(n) + 2n + 4^2T(\frac{n}{4}) = c(1 + 2)n + 4^2T(\frac{n}{4})$$
$$\leq c(1 + 2)n + 4^2[c(\frac{n}{4}) + 4T(\frac{n}{8})] = c(1 + 2 + 2^2)n + 4^3T(\frac{n}{2^3})$$
$$\ldots$$
$$\leq cn \sum_{i=0}^{k-1} 2^i + 4^kT(\frac{n}{2^k})$$
$$\ldots$$
$$\leq cn \sum_{i=0}^{\log n-1} 2^i + 4^{\log n}T(1) = cn(n - 1) + dn^2 \in O(n^2)$$

Well, that was disappointing! What can we learn?

- We need to get a feeling for what the value of a recurrence is going to be.
- We need a machine to help us solve them.
- We haven’t made much progress on integer multiplication!