CS 170 HW 1

Due on 2017-01-29, at 11:59 pm

1 (★) Study Group

List the names and SIDs of the members in your study group.

2 (★★★) Recurrence Relations

Solve the following recurrence relations and give a Θ bound for each of them.

(a) (i) $T(n) = 3T(n/4) + 4n$

(ii) $T(n) = 45T(n/3) + .1n^3$

(iii) $T(n) = T(n-1) + c^n$, where c is a constant.

(b) $T(n) = 2T(\sqrt{n}) + 3$, and $T(2) = 3$. (Hint: this means the recursion tree stops when the problem size is 2)

Solution:

(a) (i) Since $\log_4 3 < 1$, by the Master Theorem, $T(n) = \Theta(n)$.

(ii) Since $\log_3 45 > 3$, by the Master Theorem, $T(n) = \Theta(n^{\log_3 45})$.

(iii) Expanding out the recurrence, we have $T(n) = \sum_{i=0}^{n} c^i$. From the previous problem, we know that this is $\Theta(1)$, $\Theta(n)$, or $\Theta(c^n)$, depending on if $c < 1$, $c = 1$, or $c > 1$.

(b) *Solution 1:* A priori, this problem does not immediately look like it satisifies the Master's theorem because the the recurrence relation is not a map $n \to n/b$. However, we notice that we may be able to *transform* the recurrence relation into one about another variable such that it satisfies the Master's Theorem. Let k be the solution to

$$n = 2^k.$$

Then we can rewrite our recurrence relation as

$$T(2^k) = 2T(2^{k/2}) + 3.$$

Now, if we let $S(k) = T(2^k)$, then the recurrence relation becomes somethign more managable:

$$S(k) = 2S(k/2) + 3.$$

By Master's theorem, this has a solution of $\Theta(k)$. Since $n = 2^k$, then $k = \log n$ and therefore $\Theta(k) = \Theta(\log n)$.

The intuition between the transformation between n and k is that n could be a number and k, the number of bits required to represent n. The recurrence relation is easier expressed in terms of the number of bits instead of the actual numbers.

Solution 2: The recursion tree is a full binary tree of height h, where h satisfies $n^{1/2^h} = 2$. Solving this for h, we get that $h = \log\log n)$. The work done at every node of this recursion tree is constant, so the total work done is simply the number of nodes of the tree, which is $2^{h+1} - 1 = \Theta(\log n)$, so $T(n) = \Theta(\log n)$.

3 (★★★★) Majority Elements

An array $A[1 \ldots n]$ is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be **no** comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as GIF files, say.) However you *can* answer questions of the form: "is $A[i] = A[j]$?" in constant time. Four part solutions are required for each part below.

(a) Show how to solve this problem in $O(n \log n)$ time. (Hint: Split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

(b) Can you give a linear-time algorithm?

Solution:

(a) **Main idea** Divide and conquer. At each step, partition into two arrays of equal size. Notice that if A has a majority element v, v must also be a majority element of A_1 or A_2 or both.

Psuedocode Note: The recursive calls in this algorithm to smaller arrays do not require duplicating the subarray. Instead, appropriate use of indices will suffice. We express the solution here without indices for legibility.

Let array A containing n values be the input to the algorithm.

```
Majority($A$):
  (a) If $n = 1$, then return $A[1]$.
  (b) Otherwise, let $H_1, H_2$ be the two halves of the array.[1]
  (c) Let $v_i$ = Majority($H_i$) for $i = 1, 2$.
  (d) In a linear sweep over $A$, count the number of elements of $A$ equalling $v_i$. Call this $c_i$.
  (e) Return $v_i$ for any $i$ such that $c_i > n/2$. Otherwise, return $\perp$.
```

Proof of correctness By strong induction:

Induction hypothesis The algorithm is correct for $k \leq n - 1$.

Base Case If $n = 1$, the array contains exactly one element, and we always return it.

Induction step: Suppose some element v appears more than $n/2$ times in the array. Then after we partition into sub-arrays of sizes n_1, n_2, with $n_1 + n_2 = n$, either the first array contains more than $n_1/2$ copies of v, or the second array contains more than $n_2/2$ copies of v. Either way, v is a majority element of at least one of the sub-arrays, and by our induction hypothesis will be returned by the recursive call to the algorithm.

[1] For $n = 2k + 1$, break the array into pieces of size k and $k + 1$.

Running time analysis Two calls to problems of size $n/2$, and then linear time to test $v1, v2$: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

(b) **Main idea** Removing two different items does not change the majority element (but may create spurious solutions!). We recursively pair up the elements and remove all pairs with different elements. For every pair with identical elements, it suffices to keep just one (since we do so for all pairs, we again don't change the majority). If n is odd, brute-force test in linear time whether the unpaired element is a majority element.

Psuedocode

Let array A containing n values be the input to the algorithm.

$\texttt{M}\widetilde{\texttt{ajori}}\texttt{ty}(A)$: If n is odd, check if the first element is the majority element in a linear sweep over the array. If it is, return it. If not, throw it away. Instead if n is even, in a linear sweep over the array A, compare elements in pairs. If the pair of elements are same, keep one of them. If they are different, throw them both away.

Recurse on the reduced array until either one element remains or the array is empty (in which case, return $\perp$).

Proof of correctness By strong induction:

Induction hypothesis The algorithm is correct for $k \leq n - 1$.

Base Case If $n = 1$, the array contains exactly one element, and we always return it.

Induction step If n is odd, we test whether the first element is the majority element. If it is not, discarding it will certainly preserve the real majority element (if it exists). Consider now when n is even. We can seperate the process into two steps for the analysis. First, consider all pairs consisting of two distinct elements. Discarding all these pairs preserves the majority element v. This is because if we discarded m pairs, we could have discarded at most m copies of v. So we are left with at least $\frac{n}{2} - m$ copies of v among $n - 2m$ elements, and therefore v is still a majority element. Now, the only pairs that remain have matching elements. So we can choose one from each pair, we leave the proportion of every element u unchanged. Therefore, this transformation preserves the majority element but leaves us with an array of size $\frac{n}{2} - m < n$. The correctness of the recursive call to the algorithm now follows from the induction hypothesis.

Running time analysis For any n, in two recursions of the algorithm, the problem size reduces to $\leq n/2$. The overhead is $O(n)$. This follows the recursion formula,

$$T(n) = T(\frac{n}{2}) + O(n) = O(n).$$

4 (★★★★) Squaring vs multiplying: matrices

The square of a matrix A is its product with itself, AA.

(a) Show that five multiplications are sufficient to compute the square of a 2×2 matrix.

(b) What is wrong with the following algorithm for computing the square of an $n \times n$ matrix?

"Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part (a). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in $\Theta(n^{\log_2 5})$ time."

(c) In fact, squaring matrices is no easier than multiplying them. Show that if $n \times n$ matrices can be squared in $\Theta(n^c)$ time, then any $n \times n$ matrices can be multiplied in $\Theta(n^c)$ time.

Solution:

a)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a^2 + bc & b(a+d) \\ c(a+d) & bc + d^2 \end{bmatrix}$$

Hence the 5 multiplications $a^2, d^2, bc, b(a+d)$ and $c(a+d)$ suffice to compute the square.

b) We have:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^2 = \begin{bmatrix} A^2 + BC & AB + BD \\ CA + DC & CB + D^2 \end{bmatrix} \neq \begin{bmatrix} A^2 + BC & B(A+D) \\ C(A+D) & BC + D^2 \end{bmatrix}.$$

We end up getting 5 subproblems that are *not of the same type as the original problem*: We started with a squaring problem for a matrix of size $n \times n$ and three of the 5 subproblems now involve *multiplying* $n/2 \times n/2$ matrices. Hence the recurrence $T(n) = 5T(n/2) + O(n^2)$ does not make sense.

(Also, note that matrices don't commute! That is, in general $BC \neq CB$, so we cannot reuse that computation)

c) Given two $n \times n$ matrices X and Y, create the $2n \times 2n$ matrix A:

$$A = \begin{bmatrix} 0 & X \\ Y & 0 \end{bmatrix}$$

It now suffices to compute A^2, as its upper left block will contain XY:

$$A^2 = \begin{bmatrix} XY & 0 \\ 0 & YX \end{bmatrix}$$

Hence, the product XY can be calculated in time $O(S(2n))$. If $S(n) = O(n^c)$, this is also $O(n^c)$.

Note: This is an example of a reduction, and is an important concept that we will see over and over again in this course. We are saying that matrix squaring is no easier than matrix multiplication — because we can trick any program for matrix squaring to actually solve the more general problem of matrix multiplication.

5 (★★★) Hadamard matrices

The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $[1]$
- For $k > 0, H_k$ is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

(a) Write down the Hadamard matrices H_0, H_1, and H_2.

Solution: $H_0 = 1$

$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

$H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$

(b) Compute the matrix-vector product H_2v, where H_2 is the Hadamard matrix you found above, and $v = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$ is a column vector. Note that since H_2 is a 4×4 matrix, and v is a vector of length 4, the result will be a vector of length 4.

Solution:

$$H_2v = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \end{bmatrix}$$

(c) Now, we will compute another quantity. Take v_1 and v_2 to be the top and bottom halves of v respectively. Therefore, we have that $v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, $v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, and $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$. Compute $u_1 = H_1(v_1 + v_2)$ and $u_2 = H_1(v_1 - v_2)$ to get two vectors of length 2. Stack u_1 above u_2 to get a vector u of length 4. What do you notice about u?

Solution:

$$H_1(v_1 + v_2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$H_1(v_1 - v_2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

We notice that $u = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \end{bmatrix} = H_2v$

(d) Suppose that

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

is a column vector of length $n = 2^k$. v_1 and v_2 are the top and bottom half of the vector, respectively. Therefore, they are each vectors of length $\frac{n}{2} = 2^{k-1}$. Write the matrix-vector product $H_k v$ in terms of H_{k-1}, v_1, and v_2 (note that H_{k-1} is a matrix of dimension $\frac{n}{2} \times \frac{n}{2}$, or $2^{k-1} \times 2^{k-1}$). Since H_k is a $n \times n$ matrix, and v is a vector of length n, the result will be a vector of length n.

Solution: $H_k v = \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 \\ H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix} = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$

(e) Use your results from (c) to come up with a divide-and-conquer algorithm to calculate the matrix-vector product $H_k v$, and show that it can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

Solution: Compute the 2 subproblems described in part (d), and combine them as described in part (c). Let $T(n)$ represent the time taken to find $H_k v$. We need to find the vectors $v_1 + v_2$ and $v_1 - v_2$, which takes $O(n)$ time. And we need to find the matrix-vector products $H_{k-1}(v_1 + v_2)$ and $H_{k-1}(v_1 - v_2)$, which take $T(\frac{n}{2})$ time. So, the recurrence relation for the runtime is:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Using the master theorem, this give us $T(n) = O(n \log n)$.