Exercises - Divide and Conquer (1) - Algoritmiek

Tutorial February 6, 2024

1. **Bounds:** Consider a sorted array A of (not per se distinct) integers and two integers $\ell \leq r$.

   (a) Show how to decide whether an integer k is contained in A that satisfies $\ell \leq k \leq r$ in $O(\log n)$ time.

   (b) Show how to count how many integers k are contained in A that satisfy $\ell \leq k \leq r$ in $O(\log n)$ time.

   Solution:

   (a) Do a binary search for ℓ. When the search stops at index i of the array, check whether $A[i]$ or $A[i+1]$ is in range.

   (b) Do a binary search for ℓ and for r. The indices of where the searches stop will enable you to count. Are any special modifications to the binary search needed if the integers in A are not necessarily distinct? How would you implement this in C#?

2. **Master theorem:**

   (a) Consider the recurrence $T(n) = 4T(n/2) + n^3$; $T(1) = 1$. What solution does the Master theorem give?

   A. Apply Case 1 to get $\Theta(n^2)$
   B. Apply Case 2 to get $\Theta(n^2 \log n)$
   C. Apply Case 3 to get $\Theta(n^3)$
   D. The Master Theorem does not apply.

   (b) Consider the recurrence $T(n) = 0.5T(n/2) + n \log n$; $T(1) = 1$. What solution does the Master theorem give?

   A. Apply Case 1 to get $\Theta(1/n)$
   B. Apply Case 2 to get $\Theta(n \log^2 n)$
   C. Apply Case 3 to get $\Theta(n \log n)$
   D. The Master theorem does not apply.

   (c) Consider the following two recurrences: $T(n) = 2T(n/2) + O(n \log n)$ and $T(n) = 2T(n/2) + O(n\sqrt{n})$, both with $T(n) = 1$. Use the Master theorem to solve them. Are the solutions the same or different? Discuss why.

   (d) Consider the following recurrence: $T(n) = 2T(n/2) + O(2^n)$; $T(1) = 1$. Does the Master Theorem apply? Discuss why/why not.

   (e) Consider Problems 4-1 and 4-4 of the book to exercise more with the Master theorem.

   Solution:

   (a) The correct answer is C. $a = 4, b = 2$. Note $a \geq 2$ and $b > 1$ so the Master theorem applies. Calculate $p = \log_2 4 = 2$. Because $g(n) = n^3$, the first two cases do not apply (explain why). For the third case, indeed $g(n) = \Omega(n^{2+\epsilon})$ for some $\epsilon > 0$ (namely $\epsilon = 1$). Moreover, there is a constant $c < 1$ such that $a \cdot g(n/b) \leq c \cdot g(n)$; indeed, $4(n/2)^3 \leq (1/2)n^3$, so taking $c = 1/2$ suffices.

(b) The correct answer is C or D. $a = 0.5, b = 2$, but the Master theorem (book 3rd edition) assumes that $a \geq 1$. The value of p turns negative. The correct answer, which one can prove by substitution, is that $T(n) = \Theta(n \log n)$; that is, $T(n) \leq 2n \log n + 1$ can be proven (try it!), and clearly $n \log n$ is a lower bound. The Master theorem (book 4th edition) works for any $a > 0$ and leads to C, $\Theta(n \log n)$.

(c) The first recurrence solves to $T(n) = \Theta(n \log^2 n)$ by Case 2. The second to $T(n) = \Theta(n\sqrt{n})$ by Case 3 (argue that the condition on g holds!). Note that $\sqrt{n}$ is a polynomial factor, which makes a big difference in the final running time.

(d) Note that 2^n grows faster than n^e for any constant e. Hence, Case 3 of the Master theorem apply (argue that the condition on g holds!).

(e) No answers provided.

3. **Substitution:** Consider the following recurrences. Solve them (or prove a very good upper bound) using the substitution method. In all cases, $T(1) = 1$.

(a) $T(n) = T(n/2) + 1$ (what is the subtle difference compared to what we saw in class?)

(b) $T(n) = 3 \cdot T(n-1)$

(c) $T(n) = T(n-1) + n$

Solution:

(a) Guess $T(n) = 1 + \log n$.
Basis: $T(1) = 1$. $1 + \log n = 1$. Correct.
IH: for all $1 \leq n' < n$, $T(n') = 1 + \log n'$.
Step: $T(n) = T(n/2) + 1 = 2 + \log(n/2) = 2 + \log n - \log 2 = 1 + \log n$ by definition and induction.

(b) Guess: $T(n) = 3^{n-1}$. (Using $T(n) \leq 3^n$ also yields a valid solution)
Basis: $T(1) = 1 = 3^0 = 3^{1-1}$.
IH: for all $1 \leq n' < n$, $T(n') = 3^{n'-1}$.
Step: $T(n) = 3 \cdot 3^{n-2} = 3^{n-1}$ by definition and IH.

(c) Guess: $T(n) \leq cn^2 + dn + e$ for constants c, d, e to be determined.
Basis: $T(1) = 1$. $cn^2 + dn + e = c + d + e$. Hence, we need that $1 \leq c + d + e$.
IH: for all $1 \leq n' < n$, $T(n') \leq cn'^2 + dn' + e$.
Step: $T(n) = T(n-1) + n \leq c(n-1)^2 + d(n-1) + e + n = cn^2 - 2cn + c + dn - d + e + n$ by definition and IH. Hence, for $T(n) \leq cn^2 + dn + e$, we need $cn^2 + (d - 2c + 1)n + (e - d + 1) \leq cn^2 + dn + e$. It suffices if $d - 2c + 1 \leq d$ and $e - d + 1 \leq e$.
Use $c = 1$, $d = 1$, $e = 0$.

4. **Minimum finding:** Finding the minimum in an array can be seen as a divide & conquer algorithm.

(a) Give the recursive algorithm to find the minimum. Then prove using induction why the algorithm is correct.

(b) Give the recurrence for the running time of the algorithm.

(c) Solve the recurrence. Use at least two of the methods covered in the lecture.

Solution:

(a)
```
minimum(Array A, long left, long right)
{
if right-left == 1
then return A[left]
else return Math.min(minimum(A, left, (left+right)/2), \
minimum(A, (left+right)/2), right))
}
```

To prove: `minimum(A, 0, A.length)` gives the minimum of array A. Apply induction on `right-left`

Basis: `right-left = 1`. Array has size 1, answer is the only element in range en thus correct.

Hypothesis: answer is correct for all values `right-left` smaller than i for $i > 1$.

Step: `right-left` $= i$ with $i > 1$. The current range is partitioned into two parts, each smaller than the whole (note `left` $<$ `(right+left)/2` $<$ `right`). By the induction hypothesis, `minimum` works correctly for these smaller parts and delivers the minimum. By the correctness of `Math.min`, the algorithm correctly outputs the minimum of this range and the step is correct.

So indeed, `minimum` is correct.

(b) $T(n) = 2T(n/2) + c$; $T(1) = c'$ for some constants c, c'.

(c) **Substitution**: Guess: $T(n) = O(n)$ so $T(n) \leq dn + e$ for constants d, e.

Basis: $n = 1$. $T(1) = c'$, $dn + e = d + e$. So $T(1) \leq d + e$ if $d + e \geq c'$.

IH: for all $1 \leq n' < n$, $T(n') \leq dn' + e$.

Step: $T(n) = 2T(n/2) + c \leq 2(dn/2 + e) + c = dn + c + 2e$ by definition and the IH. So $T(n) \leq dn + e$ if $dn + c + 2e \leq dn + e$ and thus $c + 2e \leq e$ and thus $e \leq -c$.

Our guess is true if $e \leq -c$ and $d + e \geq c'$. Now pick $e = -c$ and $d = c' + c$.

Note that it does not work without the $+e$!

Master theorem: $a = 2$, $b = 2$, $g(n) = c$. Hence, $p = 1$. So we are in Case 1, and $T(n) = \Theta(n^p) = \Theta(n)$.

Recursion tree: It is a binary tree of depth $\log n$, so it has n nodes. We spend $O(1)$ time in every node. So the running time is $O(n)$.

5. ***O*-notation (repeat from Datastructuren):** This exercise is based on Problem 3-4 in Cormen et al. Let $f(n)$ and $g(n)$ be two non-negative functions. Answer the questions below. If your answer is yes, give a proof; if your answer is no, give a counterexample or counter-proof.

   - Is it true that $5n^2 = O(n^2)$?
   - Is it true that $5n^2 = \Theta(n^2)$?
   - Is it true that $5n^2 = o(n^2)$?
   - Does $f(n) = O(g(n))$ imply that $g(n) = O(f(n))$?
   - Does $f(n) = o(g(n))$ imply that $g(n) = \omega(f(n))$?
   - Does $f(n) = O(g(n))$ imply that $2^{f(n)} = O(2^{g(n)})$?
   - Is it true that $f(n) + o(f(n)) = \Theta(f(n))$?

   Solution:

   - Yes. $0 \leq 5n^2 \leq cn^2$ for $c = 5$ and all $n \geq n_0 = 1$.
   - Yes. $0 \leq c_1 n^2 \leq 5n^2 \leq c_2 n^2$ for $c_1 = c_2 = 5$ and all $n \geq n_0 = 1$.
   - No. There exists a positive constant $c > 0$ such that for all $n \geq n_0 > 0$ it holds that $5n^2 \not\leq cn^2$, namely $c = 4$.
   - No. For example, let $f(n) = n$ and $g(n) = n^2$.
   - Yes. This is immediate by the definition in Cormen et al.
   - No. Let $f(n) = n^2 + n$ and $g(n) = n^2$. Clearly, $f(n) = O(g(n))$. However, $2^{f(n)} = 2^{n^2+n} = 2^{n^2} \cdot 2^n$, whereas $2^{g(n)} = 2^{n^2}$. There is an extra 2^n factor in $2^{f(n)}$ that cannot be hidden in the big-O of $O(2^{n^2})$.
   - Yes. $g(n) = o(f(n))$ means that for all positive constants $c > 0$ there exists an $n_0 > 0$ such that $0 \leq g(n) < cf(n)$ for all $n \geq n_0$. Pick $c = 1$ and let n_0 correspond to this choice of c. Then $f(n) + g(n) = \Theta(f(n))$, because $0 \leq f(n) \leq f(n) + g(n) \leq 2f(n)$ for all $n_0 > 0$.

6. ***O*-notation:**

   (a) Consider the following functions and order them according to their asymptotic growth: 2^n, $\log\log n$, $\sqrt{n}$, $\log n$, n^3, $2^{\sqrt{n}}$, $\log^5 n$.

   (b) Create a table with ten columns and many (16) rows. Label the first column by 'function', the second by 'constant', and the further seven by the above functions in order of increasing asymptotic growth. The final column should be labeled 'too big'.

   (c) Consider the following functions and put them the first column in any order: $(1.01)^n$, $n/\log n$, $\sqrt{n}\log n$, $\log(n^3)$, $\sqrt{\log n}$, $\log(n\log n)$, $2^{\log n}$, 2^{n^2}, $\log 128$, $1/n$, $\log(n) \cdot \log(n)$, $2^{\sqrt{n}} + n^2$, $\log^7 n$, $\sqrt[3]{n}$, $2 \cdot 2^n$.

   (d) Consider a cell of the table. Let $f(n)$ be the function in the row and $g(n)$ be the function in the column. Put a mark in the cell if $f(n) = O(g(n))$. Use the column 'too big' if you think f is not big-Oh of any function. For example, in the row marked $n/\log n$ and column marked 2^n, there should be a mark. Fill out the entire table.

   (e) Why does it not make sense to add a column 'too small'?

   (f) Do the same for $\Omega()$ instead of $O()$. What about 'too big' versus 'too small' in this case?

   Solution:

(a) $\log\log n$, $\log n$, $\log^5 n$, $\sqrt{n}$, n^3, $2^{\sqrt{n}}$, 2^n.

(b)

(c)

(d)

function	constant	$\log\log n$	$\log n$	$\log^5 n$	$\sqrt{n}$	n^3	$2^{\sqrt{n}}$	2^n	too big
$(1.01)^n$							X	X	
$n/\log n$						X	X	X	
$\sqrt{n}\log n$						X	X	X	
$\log(n^3)$			X	X	X	X	X	X	
$\sqrt{\log n}$			X	X	X	X	X	X	
$\log(n\log n)$			X	X	X	X	X	X	
$2^{\log n}$						X	X	X	
2^{n^2}									X
$\log 128$	X	X	X	X	X	X	X	X	
$1/n$	X	X	X	X	X	X	X	X	
$\log(n)\cdot\log(n)$				X	X	X	X	X	
$2^{\sqrt{n}}+n^2$							X	X	
$\log^7 n$					X	X	X	X	
$\sqrt[3]{n}$					X	X	X	X	
$2\cdot 2^n$								X	

(e) A function that might fit this column would also be big-Oh of any constant.

(f) The table should have its marks exactly inverted (except for three cells that are marked in both tables!)

Optional, more difficult exercise.

6. **Memory Usage of MergeSort:** Suppose you implement MergeSort as follows:

```
MergeSort(A, a, b) {
if b-a==1 then return new int[] {A[a]};
R1 = MergeSort(A,a,(a+b)/2);
R2 = MergeSort(A,(a+b)/2,b);
R = new int[b-a];
merge R1 and R2 into the array R
return R; }
```

(a) How much memory do you allocate in total if you run this implementation on an array of n integers?

(b) Your input is large and therefore, you try to write an implementation of MergeSort that uses less memory. Give an implementation that allocates $n + O(\log n)$ working memory in total (or $2n + O(\log n)$ if you count the input), but still has running time $O(n \log n)$.

(c) Your input is very large and therefore, you try to write an implementation of MergeSort that uses much less memory. Give an implementation that allocates $(n/2) + O(\log n)$ working memory in total (or $(3/2)n + O(\log n)$ if you count the input), but still has running time $O(n \log n)$.
Hint: You will need, amongst others, the answer to part (b).

Solution:

(a) Create a recurrence for the amount of allocated memory. Let $M(n)$ denote the amount of memory used when $b - a = n$. Then clearly $M(1) = 1$ and $M(n) = 2M(n/2) + n$. We have seen that this recurrence solves to $M(n) = O(n \log n)$ (try to recall the proof).

(b) Create a separate 'global' array where you merge into. You can also extend your original array A by n elements, and use that extra space as 'working memory'.

(c) For the recursive call on the left side L of the array, use the right side R of the array as 'working memory'. Be careful in the merge, that when you want to place an element $L[i]$ into spot $R[j]$, that you swap $L[i]$ and $R[j]$, as not to destroy the element in $R[j]$ (which you have not sorted yet). After this step, you have effectively swapped L (sorted) and R (unsorted) in memory, that is, L occupies the right half and R the left half. Now sort R into an extra array E using the method of part (b); note that an extra array E of size $n/2$ suffices to merge into. So now E contains R (sorted) and L is sorted. It remains to merge E and L. Note that $n/2$ elements of consecutive space is left (occupied by unsorted/unmerged R). This is enough working space to do the merge (at any time, we can place the remainder of E into memory without overlapping the remainder of L).