PROGRAMMING AND ALGORITHMS II

GABRIEL VALIENTE

ALGORITHMS, BIOINFORMATICS, COMPLEXITY AND FORMAL METHODS
RESEARCH GROUP, TECHNICAL UNIVERSITY OF CATALONIA

2023–2024

ANALYSIS OF RECURSIVE ALGORITHMS

RECURRENCE RELATIONS AND THEIR SOLUTION METHODS

EXAMPLES

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 4th edition, 2022

When an algorithm contains a recursive call, you can often describe its running time by a recurrence, which describes the overall running time on a problem of size n in terms of the running time of the same algorithm on smaller inputs.

Let $T(n)$ be the worst-case running time on a problem of size n.

If the problem size is small enough, say $n < n_0$ for some constant $n_0 > 0$, the straightforward solution takes constant time, which we write as $\Theta(1)$.

Otherwise, suppose the algorithm makes a recursive calls, each on a subproblem of size n/b, that is, $1/b$ the size of the original.

It takes $T(n/b)$ time to solve one subproblem of size n/b, and so it takes $aT(n/b)$ to solve all a of them.

We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ aT(n/b) + \Theta(\ldots) & \text{otherwise} \end{cases}$$

The substitution method for solving recurrences consists of guessing the form of the solution using symbolic constants, and using mathematical induction to show that the solution works, and find the constants.

The recursion-tree method for solving recurrences consists of drawing up a recursion tree (where each node represents the cost of a single subproblem somewhere in the set of recursive function invocations) to generate intuition for a good guess, which you can then verify by the substitution method.

The master method for solving recurrences consists of memorizing three cases of recurrences of the form $T(n) = aT(n/b) + f(n)$, which describe the running time of algorithms that divide a problem of size n into a subproblems, each of size $n/b < n$, which are solved recursively, and the function $f(n)$ encompasses the cost of dividing the problem before the recursion, as well as the cost of combining the results of the recursive solutions to subproblems.

```
function LINEAR-SEARCH($A, x, low, high$)
    $i = low$
    while $i \leqslant high$ and $x \neq A[i]$ do
        $i = i + 1$
    if $i > high$ then
        return nil
    return $i$
```

```
function LINEAR-SEARCH($A, x, low, high$)
    if $low > high$ then
        return nil
    else if $x = A[low]$ then
        return $low$
    else
        return LINEAR-SEARCH($A, x, low + 1, high$)
```

Both procedures terminate the search unsuccessfully when the range is empty (that is, $low > high$) and terminate it successfully if the value x has been found.

Based on the comparison of x to the first element in the searched range, the search continues with the remaining elements in the range.

The recurrence for these procedures is therefore

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ T(n-1) + \Theta(1) & \text{otherwise} \end{cases}$$

whose solution is $\Theta(n)$.

Denote by c the constant hidden in the $\Theta(1)$ term.

Guess that $T(n) \leqslant dn$ for a constant d to be chosen. We have

$$T(n) \leqslant T(n-1) + c \leqslant d(n-1) + c = dn - d + c$$

Now, $dn - d + c \leqslant dn$ if $-d + c \leqslant 0$, which is equivalent to $d \geqslant c$, and this holds for all n.

Guess that $T(n) \geqslant dn$ for a constant d to be chosen. We have

$$T(n) \geqslant T(n-1) + c \geqslant d(n-1) + c = dn - d + c$$

Now, $dn - d + c \geqslant dn$ if $-d + c \geqslant 0$, which is equivalent to $d \leqslant c$, and this holds for all n.

Thus, $T(n) = \Theta(n)$.

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ T(n-1) + 1 & \text{if } n > 2 \end{cases}$$

is $T(n) = n$.

The base case is when $n = 2$, and we have $n = 2$.

For the inductive step, our hypothesis is that $T(n-1) = n - 1$. Then

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= n - 1 + 1 \\ &= n \end{aligned}$$

which completes the inductive proof for exact powers of 2.

```
function BINARY-SEARCH($A, x, low, high$)
    while $low \leqslant high$ do
        $mid = \lfloor (low + high)/2 \rfloor$
        if $x = A[mid]$ then
            return $mid$
        else if $x > A[mid]$ then
            $low = mid + 1$
        else
            $high = mid - 1$
    return nil
```

```
function BINARY-SEARCH($A, x, low, high$)
    if $low > high$ then
        return nil
    $m = \lfloor (low + high)/2 \rfloor$
    if $x = A[m]$ then
        return $m$
    else if $x > A[m]$ then
        return BINARY-SEARCH($A, x, m + 1, high$)
    else
        return BINARY-SEARCH($A, x, low, m - 1$)
```

Both procedures terminate the search unsuccessfully when the range is empty (that is, $low > high$) and terminate it successfully if the value x has been found.

Based on the comparison of x to the middle element in the searched range, the search continues with the range halved.

The recurrence for these procedures is therefore

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ T(n/2) + \Theta(1) & \text{otherwise} \end{cases}$$

whose solution is $\Theta(\lg n)$.

Denote by c the constant hidden in the $\Theta(1)$ term.

Guess that $T(n) \leqslant c \lg n$. We have

$$T(n) \leqslant T(n/2) + c \leqslant c \lg(n/2) + c = c \lg n - c \lg 2 + c = c \lg n$$

and this holds for all n.

Guess that $T(n) \geqslant c \lg n$. We have

$$T(n) \geqslant T(n/2) + c \geqslant c \lg(n/2) + c = c \lg n - c \lg 2 + c = c \lg n$$

and this holds for all n.

Thus, $T(n) = \Theta(\lg n)$.

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ T(n/2) + 1 & \text{if } n > 2 \end{cases}$$

is $T(n) = \lg n$.

The base case is when $n = 2$, and we have $\lg n = \lg 2 = 1$.

For the inductive step, our hypothesis is that $T(n/2) = \lg(n/2)$. Then

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= \lg(n/2) + 1 \\ &= (\lg n - \lg 2) + 1 \\ &= \lg n - 1 + 1 \\ &= \lg n \end{aligned}$$

which completes the inductive proof for exact powers of 2.

The worst-case behavior for quicksort occurs when the partitioning produces one subproblem with $n-1$ elements and one with 0 elements.

Let us assume that this unbalanced partitioning arises in each recursive call.

```
function PARTITION($A, p, r$)
    $x = A[r]$
    $i = p - 1$
    for $j = p$ to $r - 1$ do
        if $A[j] \leqslant x$ then
            $i = i + 1$
            exchange $A[i]$ with $A[j]$
    exchange $A[i+1]$ with $A[r]$
    return $i + 1$
```

The running time of PARTITION on a subarray $A[p:r]$ of $n = r - p + 1$ elements is $\Theta(n)$, because the **for** loop iterates $n-1$ times, each iteration takes $\Theta(1)$ time, and the parts of the procedure outside the loop take $\Theta(1)$ time, for a total of $\Theta(n)$ time.

Since the recursive call on an array of size 0 just returns without doing anything, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ T(n-1) + \Theta(n) & \text{otherwise} \end{cases}$$

whose solution is $T(n) = \Theta(n^2)$.

Denote by c the constant hidden in the $\Theta(n)$ term.

Guess that $T(n) \leqslant dn^2$ for a constant d to be chosen. We have

$$T(n) \leqslant T(n-1) + cn \leqslant d(n-1)^2 + cn = dn^2 - 2dn + d + cn$$

Now, $dn^2 - 2dn + d + cn \leqslant dn^2$ if $-2dn + d + cn \leqslant 0$, which is equivalent to $d \geqslant cn/(2n-1)$, and this holds for all $n \geqslant 1$ and $d \geqslant c$.

Guess that $T(n) \geqslant dn^2$ for a constant d to be chosen. We have

$$T(n) \geqslant T(n-1) + cn \geqslant d(n-1)^2 + cn = dn^2 - 2dn + d + cn$$

Now, $dn^2 - 2dn + d + cn \geqslant dn^2$ if $-2dn + d + cn \geqslant 0$, which is equivalent to $d \leqslant cn/(2n-1)$, and this holds for all $n \geqslant 1$ and $d \leqslant c/2$.

Thus, $T(n) = \Theta(n^2)$.

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 1 \\ T(n-1) + n - 1 & \text{if } n > 1 \end{cases}$$

is $T(n) = (n-1)n/2 + 2$.

The base case is when $n = 1$, and we have $(n-1)n/2 + 2 = 0 + 2 = 2$.

For the inductive step, our hypothesis is that $T(n-1) = (n-2)(n-1)/2 + 2$. Then

$$\begin{aligned} T(n) &= T(n-1) + n - 1 \\ &= (n-2)(n-1)/2 + 2 + n - 1 \\ &= n(n-1)/2 - 2(n-1)/2 + 2 + n - 1 \\ &= n(n-1)/2 - (n-1) + 2 + (n-1) \\ &= (n-1)n/2 + 2 \end{aligned}$$

which completes the inductive proof for exact powers of 2.

Let $T(n)$ be the worst-case running time of merge sort on n numbers.

Since the MERGE procedure on an n-element subarray takes $\Theta(n)$ time, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

whose solution is $T(n) = \Theta(n \lg n)$.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that each of the assignment statements takes constant time, and the **for** loops take $\Theta(n_L + n_R) = \Theta(n)$ time.

To account for the three **while** loops, observe that each iteration of these loops copies exactly one value from L or R back into A and that every value is copied back into A exactly once.

Therefore, these three loops together make a total of n iterations.

Since each iteration of each of the three loops takes constant time, the total time spent on these three loops is $\Theta(n)$.

To see that the solution is $T(n) = \Theta(n \lg n)$, assume that n is an exact power of 2 and that the base case is $n = 1$.

Then the recurrence is essentially

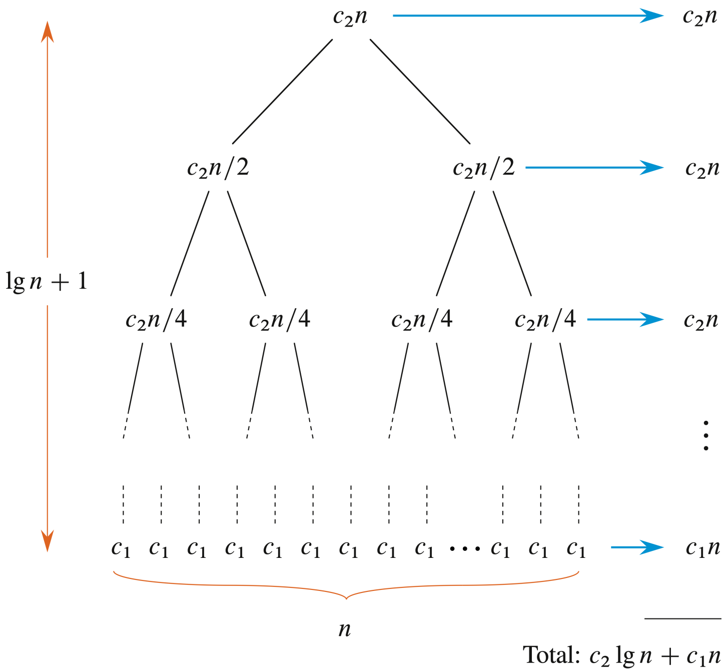
$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{if } n > 1 \end{cases}$$

where the constant $c_1 > 0$ represents the time required to solve a problem of size 1, and $c_2 > 0$ is the time per array element of the nonrecursive steps.

We can figure out the solution by means of an equivalent recursion tree representing the recurrence, where each node in the tree is expanded by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1.

The total number of levels of the recursion tree is $\lg n + 1$, where n is the number of leaves, corresponding to the input size.

The levels above the leaves each cost $c_2 n$, and the leaf level costs $c_1 n$, for a total cost of $c_2 n \lg n + c_1 n = \Theta(n \lg n)$.



$c_2 n$
$c_2 n/2$
$c_2 n/2$
$c_2 n/4$
$c_2 n/4$
$c_2 n/4$
$c_2 n/4$
c_1 c_1 c_1 c_1 c_1 c_1 c_1 c_1 c_1 $\cdots$ c_1 c_1 c_1
$\lg n + 1$
n
$c_2 n$
$c_2 n$
$c_2 n$
$c_1 n$
Total: $c_2 \lg n + c_1 n$

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

is $T(n) = n \lg n$.

The base case is when $n = 2$, and we have $n \lg n = 2 \lg 2 = 2 \cdot 1 = 2$.

For the inductive step, our hypothesis is that $T(n/2) = (n/2) \lg(n/2)$. Then

$$\begin{array}{rcl} T(n) & = & 2T(n/2) + n \\ & = & 2(n/2) \lg(n/2) + n \\ & = & n(\lg n - \lg 2) + n \\ & = & n(\lg n - 1) + n \\ & = & n \lg n - n + n \\ & = & n \lg n \end{array}$$

which completes the inductive proof for exact powers of 2.

PARAMETER PASSING

- We assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed.
- This assumption is valid in most systems because a pointer to the array is passed, not the array itself.
- CLRS Problem 4-2 examines the implications of three parameter-passing strategies:
  - An array is passed by pointer. Time $O(1)$.
  - An array is passed by copying. Time $O(N)$, where N is the size of the array.
  - An array is passed by copying only the subrange that might be accessed by the called procedure. Time $O(high - low + 1)$ if the subarray $low..high$ is passed.
- Consider the recursive binary search algorithm for finding a number in a sorted array.
  - Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences.
  - Let N be the size of the original problem and n be the size of a subproblem.
- Redo the previous part for the MERGE-SORT algorithm.

- Consider the recursive binary search algorithm for finding a number in a sorted array.
    - Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences.
    - Let N be the size of the original problem and n be the size of a subproblem.

```
function BINARY-SEARCH($A, x, low, high$)
    if $low > high$ then
        return nil
    $m = \lfloor (low + high)/2 \rfloor$
    if $x = A[m]$ then
        return $m$
    else if $x > A[m]$ then
        return BINARY-SEARCH($A, x, m + 1, high$)
    else
        return BINARY-SEARCH($A, x, low, m - 1$)
```

- $T(n) = T(n/2) + O(1)$ $\qquad T(N) = O(\lg N)$
- $T(n) = T(n/2) + O(N)$ $\qquad T(N) = O(N \lg N)$
- $T(n) = T(n/2) + O(n/2)$ $\qquad T(N) = O(N)$

- Consider the recursive merge sort algorithm for sorting an array.
  - Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences.
  - Let N be the size of the original problem and n be the size of a subproblem.

```
procedure MERGE-SORT($A$)
    MERGE-SORT($A, 1, n$)
procedure MERGE-SORT($A, p, r$)
    if $p \geqslant r$ then
        return
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q + 1, r$)
    MERGE($A, p, q, r$)
```

- $T(n) = 2T(n/2) + O(n)$ — $T(N) = O(N \lg N)$
- $T(n) = 2T(n/2) + O(n) + O(N)$ — $T(N) = O(N^2)$
- $T(n) = 2T(n/2) + O(n) + O(n/2)$ — $T(N) = O(N \lg N)$