Runtime of an algorithm

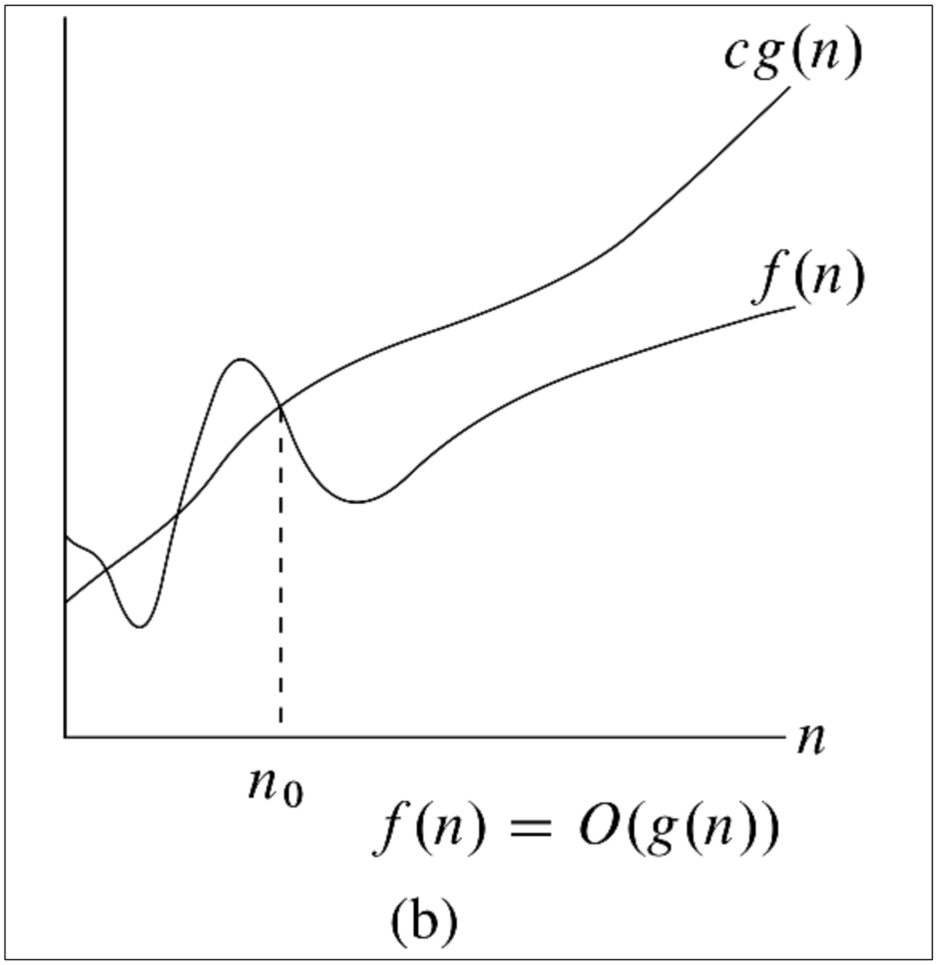
When we analyze the runtime of algorithms, the main function that we concerned with is $T(n)$, which represents the amount of time it takes to process n data items.

In many cases computing $T(n)$ directly is too hard, so we instead seek to place bounds on $T(n)$.

Big-O Notation

A function $f(n)$ is said to be $O(g(n))$ if there exists a constant C and an $N_0 > 0$ such that $f(n) \leq C\, g(n)$ for all $n \geq N_0$.

Here is a picture from the textbook that illustrates the key ideas in this definition.



The main idea here is that we want to use a simple function, $g(n)$, to put a tight upper bound on the growth of some more complex function, $f(n)$.

In addition to the O bound, the book also uses Ω and Θ bounds.

A function $f(n)$ is said to be $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$.

Analyzing the run time for a recursive algorithm

Here is pseudocode for merge sort.

```
MERGE-SORT($A, p, r$)
  if $p < r$                                  // check for base case
      $q = \lfloor (p + r)/2 \rfloor$                     // divide
      MERGE-SORT($A, p, q$)                   // conquer
      MERGE-SORT($A, q + 1, r$)               // conquer
      MERGE($A, p, q, r$)                     // combine
```

```
MERGE($A, p, q, r$)
  $n_1 = q - p + 1$
  $n_2 = r - q$
  let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
  for $i = 1$ to $n_1$
      $L[i] = A[p + i - 1]$
  for $j = 1$ to $n_2$
      $R[j] = A[q + j]$
  $L[n_1 + 1] = \infty$
  $R[n_2 + 1] = \infty$
  $i = 1$
  $j = 1$
  for $k = p$ to $r$
      if $L[i] \le R[j]$
          $A[k] = L[i]$
          $i = i + 1$
      else $A[k] = R[j]$
          $j = j + 1$
```

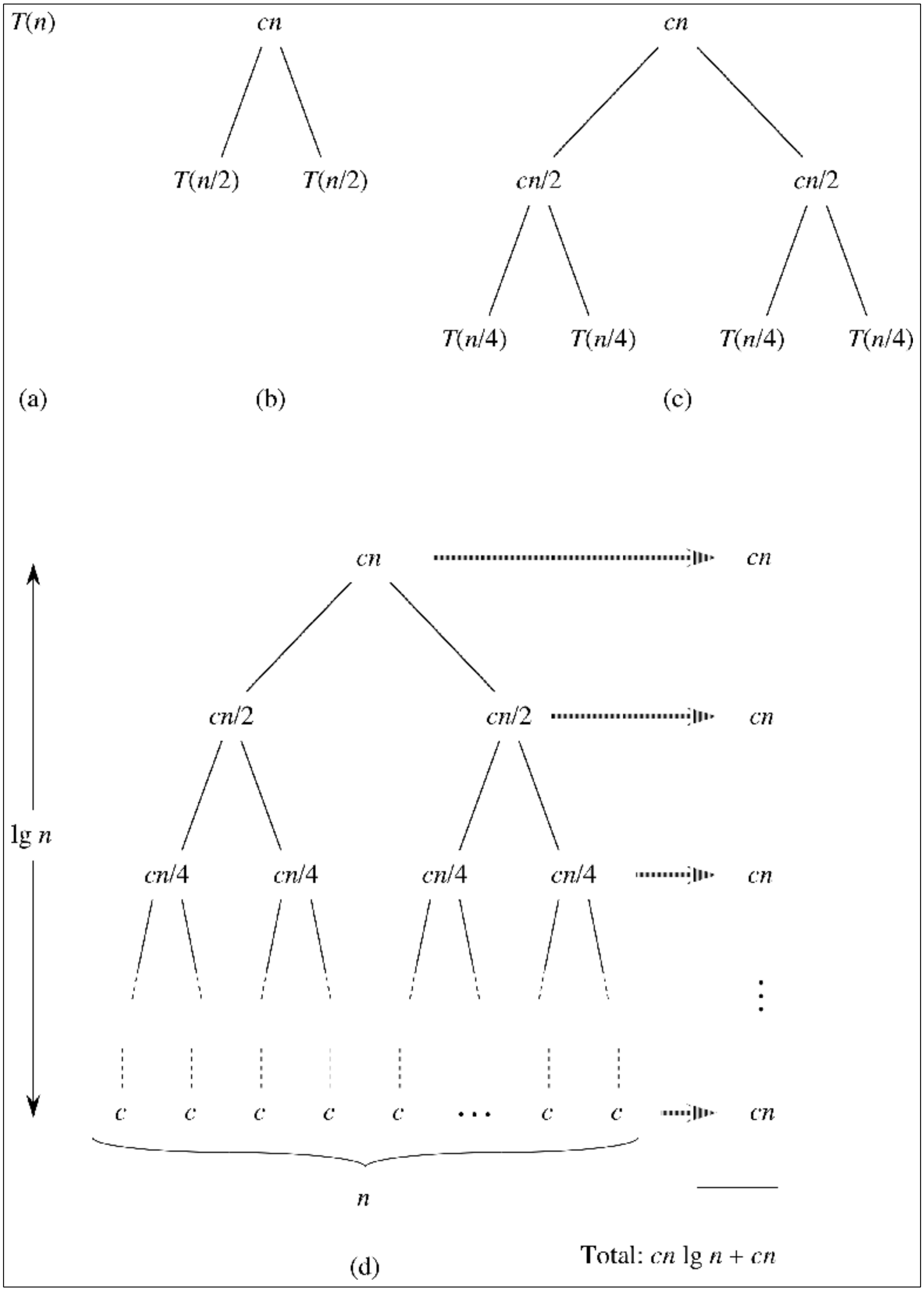
Because merge sort is a divide and conquer type of algorithm, it does its work by splitting the list to be sorted into halves and then calling itself recursively on both halves. This naturally leads to a *recurrence relation* for the run time of merge sort:

$$T(N) = T_{MERGE}(N) + 2\,T(N/2)$$

It is an easy exercise to determine that the run time for the MERGE function is $O(N)$. This makes the recurrence relation for the full merge sort

$$T(N) = c\,N + 2\,T(N/2)$$

The picture below shows one method we can use to analyze this recurrence relation, the *call tree method*. The picture shows an expanded version of the call tree for merge sort, annotated with the time used by the merges in each recursive call.



$T(n)$
cn
$T(n/2)$
$T(n/2)$
cn
$cn/2$
$cn/2$
$T(n/4)$
$T(n/4)$
$T(n/4)$
$T(n/4)$
(a)
(b)
(c)
$\lg n$
cn
$cn/2$
$cn/2$
$cn/4$
$cn/4$
$cn/4$
$cn/4$
c c c c c … c c
n
cn
cn
cn
cn
Total: $cn \lg n + cn$
(d)

An interesting observation about the time used by the merges at each level of the call tree is that they sum to $c\ n$ across the nodes in each level of the call tree. Thus, the total time used by all of the merges throughout merge sort is just $c\ n$ times the number of levels in the call tree, which is bounded by $lg\ n$. This tells us that

$$T(N) \leq c\ N\ lg\ N$$

which tells us that merge sort is $O\left(N\ lg\ N\right)$.

The substitution method

The call tree method is the most complete way to analyze the runtime of a recursive algorithm. The main drawback to the call tree method is the fact that some call trees are very complex and hard to analyze. An alternative to using a call tree is the *substitution method*.

The main idea in the substitution method is to make an educated guess for a $g(n)$ and then do an induction proof to show that $T(N)$ is $O(g(N))$. In the induction proof we use the recurrence relation for $T(N)$ along with the induction hypothesis that $T(n) \leq c\ g(n)$ for all $n < N$ and some appropriate constant c. If these two things together allow us to show that $T(N) \leq c\ g(N)$, we will have proved the induction step.

Here is an example. To use the substitution method to show that merge sort is $O(N\ lg\ N)$ we use the recurrence relation for $T(N)$:

$$T(N) = c_1\ N + 2\ T(N/2)$$

along with the induction hypothesis that

$$T(n) \leq c_2\ n\ lg\ n \text{ for all } n < N$$

to get

$$T(N) \leq c_1\ N + 2\ \left(c_2\ \left(N/2\right)\ lg\ \left(N/2\right)\right) = c_1\ N + c_2\ N\ lg\ N - c_2\ N\ lg\ 2$$

$$= c_1\ N + c_2\ N\ lg\ N - c_2\ N \leq c_2\ N\ lg\ N$$

The last inequality is true provided we choose $c_2 > c_1$.

The substitution method relies on our making a good guess for $g(n)$. Suppose we had instead guessed that $T(n)$ for merge sort was bounded above by $c_2\ n$ for all $n < N$. In that case we would have

$$T(N) \leq c_1\ N + 2\ \left(c_2\ \left(N/2\right)\right)\ = \left(c_1 + c_2\right) N$$

which is not what we want. The other thing that can go wrong in the substitution method is that we can guess a $g(n)$ that produces an upper bound which is too loose. For example, in merge sort we could have guessed that $T(n)$ is bounded by $c_2\ n^2$ for all $n < N$. This would give us

$$T(N) \leq c_1\ N + 2\ \left(c_2\ \left(N/2\right)^2\right) = c_1\ N + c_2/2\ N^2 \leq c_2\ N^2$$

which proves the bound, but is ultimately not helpful because the upper bound is too loose.

The Master Theorem

The merge sort example above showed that you can use a call tree to compute a bound for a $T(N)$ that satisfies a simple recurrence relation.

A more convenient way to develop bounds for $T(N)$ is the *master theorem for recurrences*:

Let $a \geq 1$ and $b \geq 1$ be constants and let $T(n)$ be defined by the recurrence

$$T(n) = a\ T(n/b) + f(n)$$

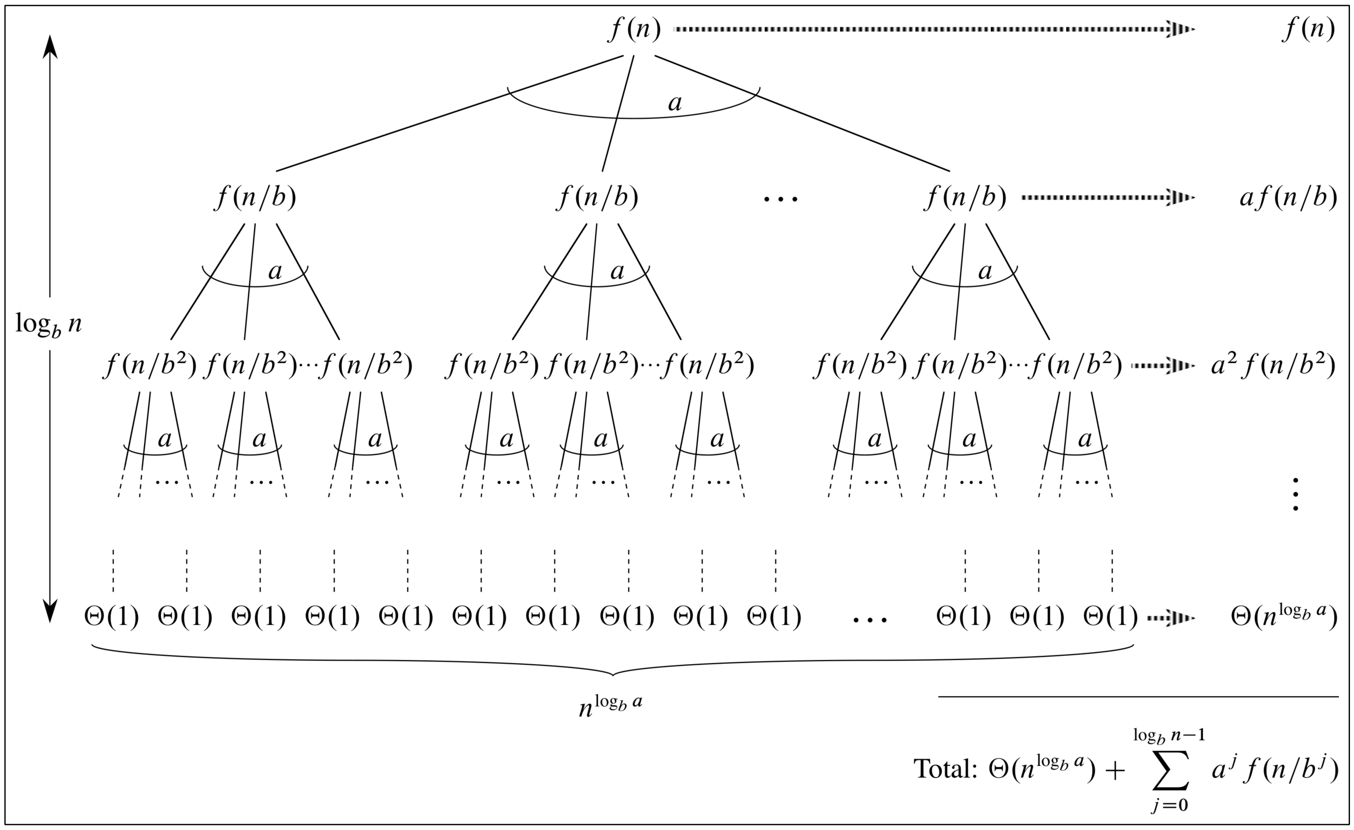
Then

1. If $f(n) = O(n^{log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.
2. If $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a}\ lg\ n)$
3. If $f(n) = \Omega(n^{log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $a\ f(n/b) < c\ f(n)$ for some $c < 1$ and all n sufficiently large, then $T(n) = \Theta(f(n))$

Ideas behind the master theorem

Here is a picture of the recursive call tree that corresponds to the recurrence

$$T(n) = a\ T(n/b) + f(n)$$



We can make the following observations about this call tree:

1. The height of the call tree is $log_b\ n$.
2. The number of nodes at level j is a^j.
3. The number of leaves is $a^{log_b n} = \left(b^{log_b a}\right)^{log_b n} = \left(b^{log_b n}\right)^{log_b a} = n^{log_b a}$. (This is known as the *watershed function*.)
4. The additional cost associated with the root node is $f(n)$.
5. The additional cost associated with each leaf node is $O(1)$.
6. The cost for all of the leaf nodes is $O(1)\ n^{log_b a} = O(n^{log_b a})$.
7. The total cost of the tree will be dominated either by the root cost, $f(n)$, or by the cost of the leaves, $n^{log_b a}$, whichever is larger. The only tricky case (case 2), happens when these two factors are comparable in size. In that case, the total cost for the tree is given by

$$\sum_{j=0}^{log_b n} f(n/b^j)\ a^j = \sum_{j=0}^{log_b n} (n/b^j)^{log_b a}\ a^j = n^{log_b a} \sum_{j=0}^{log_b n} \frac{a^j}{(b^j)^{log_b a}}$$

$$= n^{log_b a} \sum_{j=0}^{log_b n} \frac{a^j}{a^j} = n^{log_b a}\ (log_b\ n)$$

Practice problems

Use the master theorem to computer bounds for the runtime of algorithms that satisfy the follow recurrences:

- $T(N) = T(N/2) + 1$
- $T(N) = 2\ T(N/2) + n^2$