Recurrence relations

Define a function by an expression involving the same function.

Example:

$$F(n) = F(n-1) + F(n-2), \quad F(1) = F(2) = 1$$

Recurrence relations appear frequently in the analysis of algorithms.

To solve them:

- Guessing
  - † Substitution method
  - † Recursion-tree method
- General solutions
  - † Iterative method
  - † Master method

- Substitution method

  We guess a solution and then use mathematical induction to prove that our guess is correct.

- Recursion-tree method

  We convert the recurrence into a tree whose nodes represent the costs of recursive calls at various level. we then solve the recurrence by first bounding the cost of each level and then bounding the sum of all levels.

- Iterative method

  We iteratively use the recurrence to directly discover a solution.

- Master method

  We use a general solution for a class of recurrences to solve our recurrence.

Intelligent guesses

Guessing may seem like a nonscientific method, but it works well for a wide class of recurrence relations.

It works even better when we are trying to find only an upper bound ($O(\ )$ notation).

Proving a certain bound is valid is easier than computing the bound.

Given $T(n)$ satisfying the inequality

$$T(2n) \leq 2T(n) + 2n - 1, \quad T(2) = 1.$$

We want to find $f(n)$ such that

- $T(n) = O(f(n))$.
- $f(n)$ is not too far from the actual $T(n)$.

Note that this recurrence is an inequality. This is consistent with our goal of finding an upper bound and the right-hand side represents the worst case.

Try n^2

$1 = T(2) \leq 2^2 = 4$

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1 \\ &\leq 2n^2 + 2n - 1 \\ &\leq 2n^2 + 2n^2 \\ &= 4n^2 = (2n)^2 \end{aligned}$$

$T(n) = O(n^2)$

The gap from $2n^2 + 2n - 1$ to $4n^2$ is too large.

Conclusion: n^2 is probably too large!

Try $T(n) \leq cn$

$1 = T(2) \leq c \cdot 2 \Longrightarrow c > 1/2$

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1 \\ &\leq 2cn + 2n - 1 \\ &= c2n + 2n - 1 \\ &\not\leq c2n \end{aligned}$$

Conclusion: cn is too small.

Try $n\log_2 n$

$1 = T(2) < 2\log_2 2$

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1 \\ &\leq 2n\log_2 n + 2n - 1 \\ &= 2n(\log_2 n + 1) - 1 \\ &= 2n\log_2 2n - 1 \\ &< 2n\log_2 2n \end{aligned}$$

$n\log_2 n$ is very close

$T(n) = O(n\log_2 n)$

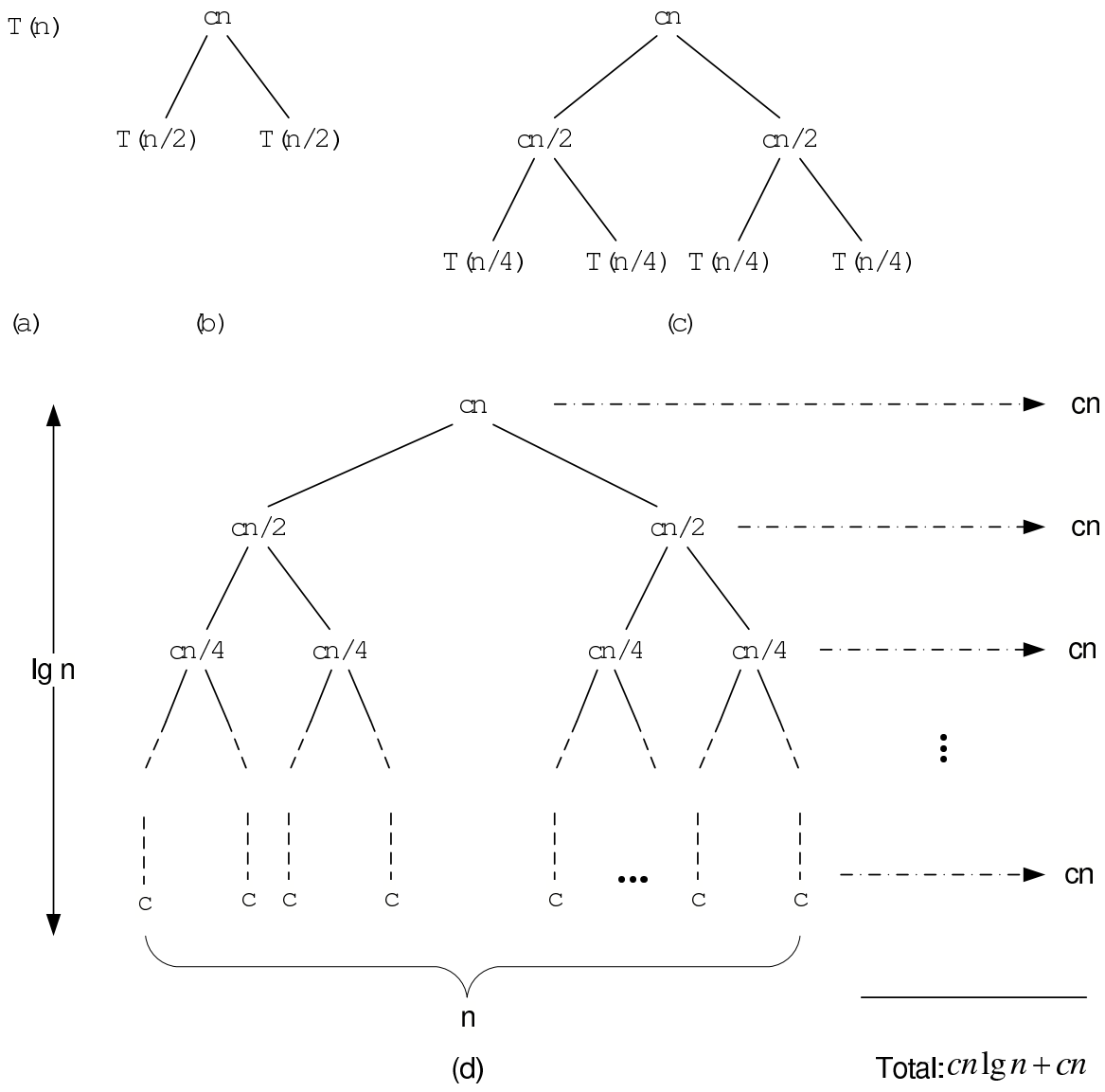
The recursion-tree method

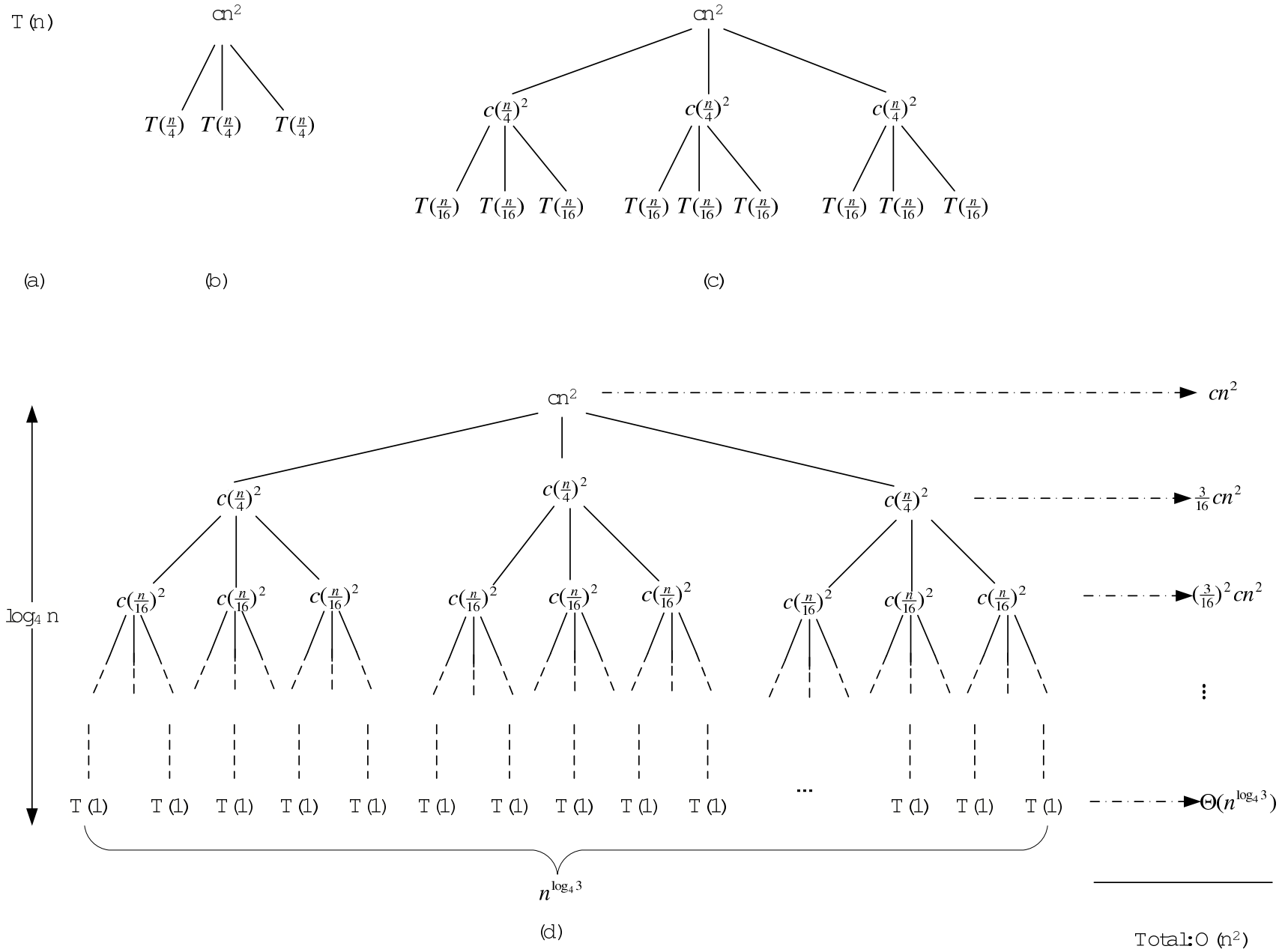
Guessing more intelligently

- Draw a recursion tree.
- Each node represent the cost of a single subprogram in the set of recursive function calls.
- Sum the costs within each level of the tree to get a set of per-level costs.
- Sum all the the per-level costs to determine the total cost of all levels of the recursion.
- Best used to generate a good guess which can then be verified by the substitution method.
- most useful when the recurrence is the running time of a divide-and-conquer algorithm.

Example: merge sort

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &= 4T(n/4) + cn + cn \\ &= 8T(n/8) + cn + cn + cn \\ &= 2^i T(n/2^i) + i \cdot cn \\ &= \cdots \\ &= cn + cn \log_2(n) \quad (\ i = \log_2(n)\ ) \end{aligned}$$


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
n
(d)
Total: $cn\lg n + cn$


T(n)
cn²
T(n/4) T(n/4) T(n/4)
(a)
(b)
c(n/4)²
T(n/16)
(c)
log₄ n
cn²
cn²
(3/16)cn²
(3/16)²cn²
T(1)
Θ(n^log₄3)
n^log₄3
(d)
Total: O(n²)

Common mistakes:

Guess the solution is $O(f(n))$ and substitute $O(f(n))$ for $T(n)$.

- The $O(\ )$ notation cannot be used in this way!

- We cannot change the constant with the change of n!

- We do not care about constants in the end, but we cannot ignore them throughout the proof.

Example:

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1 \\ &\leq 2O(n) + 2n - 1 \\ &= O(n) \quad Wrong! \end{aligned}$$

Solution: include the constants explicitly.

Recursion Example

Fibonacci numbers

$$F_i = 1 \qquad \text{if } i = 1 \text{ or } i = 2$$
$$F_i = F_{i-1} + F_{i-2} \qquad \text{if } i > 2$$

(undefined if $i \leq 0$)

A function (program) that computes Fibonacci numbers:

```
function fib(n: integer): integer;
begin
    if (n = 2) or (n = 1)
        then fib := 1
        else fib := fib (n − 1)+ fib (n − 2);
end;
```

Easy to show the correctness.

Theorem: $\text{fib}(n) = F_n$

Proof.

(by induction on n)

Induction base: $n = 1$ or $n = 2$: obvious.

Induction hypothesis: $\text{fib}(n) = F_n$ for $n \leq k$.

$\text{fib}(k+1) = \text{fib}(k) + \text{fib}\ (k-1) =$
$F_k + F_{k-1} = F_{k+1}$. $\square$

An example of iterative method

An upper bound for the fib function

Let $T(n)$ be the running time for fib(n)

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

D: constant

We iteratively use the recurrence to substitute the smaller terms.

$$
\begin{aligned}
T(n) &\leq T(n-1) + T(n-2) + D \\
&\leq 2T(n-2) + T(n-3) + 2D \\
&\leq 3T(n-3) + 2T(n-4) + 4D \\
&\leq 5T(n-4) + 3T(n-5) + 7D \\
&\dots \\
&\leq F_i T(n-i+1) + F_{i-1} T(n-i) + (F_{i+1} - 1)D \\
&\dots \\
&\leq F_{n-1} T(2) + F_{n-2} T(1) + (F_n - 1)D \\
&\leq F_n D + (F_n - 1)D \\
&= 2DF_n - D
\end{aligned}
$$

We can now prove $T(n) \leq 2DF_n - D$ by induction!

$T(n) = O(F_n)$ ignore constant!

A lower bound for fib function

Let $d \geq 1$

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

Theorem. $T(n) \geq F_n$.

Induction base: $T(1) \geq F_1$, $T(2) \geq F_2$.

Induction hypothesis: $T(n-1) \geq F_{n-1}$.

Induction step:

$$\begin{aligned} T(n) &\geq T(n-1) + T(n-2) + d \\ &\geq F_{n-1} + F_{n-2} + d \\ &\geq F_n \end{aligned}$$

Conclusions: $T(n) = \Omega(F_n)$ and $T(n) = \Theta(F_n)$.

How about F_n itself?

$$F_N = F_{N-1} + F_{N-2}, \quad N > 1, \quad F_1 = F_2 = 1$$

Solving by guesses

Guess: F doubles each time??

$$F_N = c \cdot 2^N?$$
$$c \cdot 2^N = c2^{N-1} + c2^{N-2}?$$

Impossible!

Left side is greater than right side! (c does not matter here!)

Conclusion: $F_N \neq c \cdot 2^N$

Try something else.
Try another exponential function with a smaller base!

$$\boxed{\text{Try } F_N = A^N, \quad A \text{ constant} < 2!}$$

Find A!

$A^N = A^{N-1} + A^{N-2} \Longrightarrow$
$A^2 = A + 1$, or $A^2 - A - 1 = 0$.

Remember high school stuff! $(ax^2 + bx + c = 0)$

$$A = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where $a = 1, b = -1, c = -1$.

Solutions: $A = (1 \pm \sqrt{5})/2$.

Let $A_1 = (1 + \sqrt{5})/2$ and $A_2 = (1 - \sqrt{5})/2$.

Conclusion: $F_N = \Theta(A_1^N)$ (since $A_1^N/2 \leq F_N \leq A_1^N$.)

A general solution

Linear combination of A_1^N and A_2^N:

$c_1 A_1^N + c_2 A_2^N$.

We need to use the initial condition $F_1 = F_2 = 1$ to determine c_1 and c_2.

$$F_1 = 1 \Longrightarrow c_1 \cdot (1+\sqrt{5})/2 + c_2 \cdot (1-\sqrt{5})/2 = 1$$
$$F_2 = 1 \Longrightarrow c_1 \cdot [(1+\sqrt{5})/2]^2 + c_2 \cdot [(1-\sqrt{5})/2]^2 = 1$$

Solutions: $c_1 = 1/\sqrt{5}$ and $c_2 = -1/\sqrt{5}$

Conclusion:

$$F_N = \frac{1}{\sqrt{5}} \cdot [(1+\sqrt{5})/2]^N - \frac{1}{\sqrt{5}} \cdot [(1-\sqrt{5})/2]^N.$$

Master method

A general solution for a large class of recurrences

- A problem of size n can be divided into "a" subproblems of size n/b.
- Dividing and combining the solutions of subproblems runs in time $c \cdot d(n)$.
- a, b, c are constants such that $a \geq 1$ and $b > 1$.
- $d(n)$ is a multiplicative function (a stronger requirement, please check textbook),

$$d(xy) = d(x) \cdot d(y)$$

for example, $d(n) = n^k$: $d(xy) = (xy)^k = x^k y^k = d(x) \cdot d(y)$.

- $T(n) = aT(n/b) + cd(n), \quad T(1) = c$

$$T(n) = aT(n/b) + cd(n), \quad T(1) = c$$

$$\begin{aligned}
T(n) &= aT(n/b) + cd(n), \quad T(1) = c \\
&= a[aT(n/b^2) + cd(n/b)] + cd(n) \\
&= a^2T(n/b^2) + cad(n/b) + cd(n) \\
&= a^2[aT(n/b^3) + cd(n/b^2)] + cad(n/b) + cd(n) \\
&= a^3T(n/b^3) + ca^2d(n/b^2) + cad(n/b) + cd(n) \\
&\cdots \\
&= a^iT(n/b^i) + c\sum_{j=0}^{i-1} a^jd(n/b^j)
\end{aligned}$$

(Can also use recurrence tree method)

Assume $n = b^k$.

$$\begin{aligned}
T(n) &= ca^k + c\sum_{j=0}^{k-1} a^j d(b^{k-j}) \\
&= c\sum_{j=0}^{k} a^j d(b^{k-j}) \quad (d(1) = 1) \\
&= ca^k \sum_{j=0}^{k} a^{j-k} d(b^{k-j}) \\
&= ca^k \sum_{j=0}^{k} \frac{d(b)^{k-j}}{a^{k-j}} \quad (d(b^{k-j}) = d(b)^{k-j}) \\
&= ca^k \sum_{j=0}^{k} (d(b)/a)^{k-j} \\
&= ca^k \sum_{j=0}^{k} (d(b)/a)^{j} \quad (1**)
\end{aligned}$$

Similarly, $T(n) = cd(b)^k \sum_{j=0}^{k} (a/d(b))^j \quad (2**)$.

1. If $a > d(b)$,

   From $(1 * *)$, $T(n) = O(a^k)$. Since $k = \log_b n$, $a^k = a^{\log_{b^n}} = n^{\log_{b^a}}$,

   $T(n) = O(n^{\log_{b^a}})$.

2. If $a < d(b)$,

   From $(2 * *)$, then $T(n) = O(d(b)^k)$.

   $T(n) = O(n^{\log_{b^{d(b)}}})$.

3. If $a = d(b)$,

   $T(n) = O(ka^k) = O(\log_b n \cdot n^{\log_{b^a}}) = O(n^{\log_{b^a}} \cdot \log_b n)$

Solution

Problem: $T(n) = aT(n/b) + cd(n), \quad T(1) = c$

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > d(b) \\ O(n^{\log_b a} \cdot \log_b n) & \text{if } a = d(b) \\ (\ or O(n^{\log_{b^{d(b)}} \log_b n})\ ) & \\ O(n^{\log_{b^{d(b)}}}) & \text{if } a < d(b) \end{cases}$$

Example: $d(n) = n^L$

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^L \\ O(n^L \log_b n) & \text{if } a = b^L \\ O(n^L) & \text{if } a < b^L \end{cases}$$

Example: mergesort

$$T(n) = 2T(n/2) + cn$$

$$a = b = 2, \quad L = 1$$

$a = b^L = b^1$

$$T(n) = O(n^L \log_b n) = O(n \log_2 n)$$