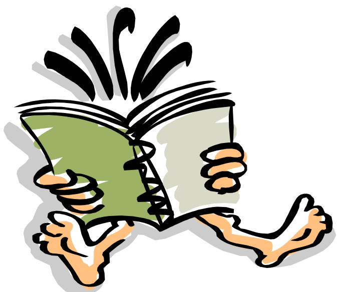
Analysis of Algorithms

Recurrences



Readings

Recurrence relations are mathematical equations that describe the **time complexity** of a recursive algorithm. They express the running time for a problem of size n, denoted as $T(n)$, in terms of the running time for smaller inputs. This is a fundamental concept in algorithm analysis, particularly for **divide and conquer** algorithms.

The Structure of a Recurrence Relation

A typical recurrence relation for a recursive algorithm consists of two parts:

1. **The Base Case:** This defines the running time for a small input size, where the recursion stops. For example, if an algorithm processes an array of size 1, it might take a constant amount of time, say $T(1) = c$.
2. **The Recursive Step:** This describes how the problem is broken down. It's usually a combination of:
   - The number of subproblems.
   - The size of each subproblem.
   - The time required to divide the problem and combine the results.

Readings

The general form for a divide and conquer recurrence is often expressed as:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- $T(n)$: The total time for a problem of size n.
- a: The number of recursive calls made by the algorithm.
- n/b: The size of each subproblem.
- $f(n)$: The time complexity of the work done outside the recursive calls (i.e., the time to divide the problem and merge the results).

Examples of Recurrence Relations

- **Binary Search:** This algorithm divides a sorted array in half and searches one of the halves. The work done is a single comparison. Therefore, its recurrence relation is:

$$T(n) = T(\frac{n}{2}) + O(1)$$

The solution to this is $T(n) = O(\log n)$.

Readings

- **Merge Sort:** This algorithm splits an array into two halves, recursively sorts each half, and then merges the two sorted halves. The merging step takes linear time, $O(n)$. Its recurrence relation is:

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

  The solution is $T(n) = O(n \log n)$.

- **Tower of Hanoi:** This is a classic puzzle with a clear recursive solution. To move n disks from one peg to another, you must:

  1. Move $n - 1$ disks to an auxiliary peg. ($T(n-1)$)
  2. Move the largest disk to the destination peg. ($O(1)$)
  3. Move the $n - 1$ disks from the auxiliary peg to the destination. ($T(n-1)$)
     Its recurrence relation is:

$$T(n) = 2T(n-1) + 1$$

     The solution to this is $T(n) = O(2^n)$.

Readings

Methods for Solving Recurrence Relations

Solving a recurrence relation means finding a "closed-form" expression for $T(n)$ that doesn't rely on itself. The three primary methods for solving them are:

1. **Substitution Method:** This involves guessing the solution and then using mathematical induction to prove that the guess is correct. It's useful when you have a good intuition for the answer.
2. **Recurrence Tree Method:** This is a visual approach where you draw a tree to represent the recursive calls. Each node represents the cost of a subproblem. You sum the costs at each level of the tree and then sum all the levels to find the total time complexity.
3. **Master Theorem:** This is a powerful, "cookbook" method for solving a specific class of recurrence relations of the form $T(n) = aT(\frac{n}{b}) + f(n)$. It provides a quick way to find the asymptotic complexity by comparing the cost of the work done in the base cases ($n^{\log_b a}$) with the cost of the work done at each level ($f(n)$).

Recurrences and Running Time

- An equation or inequality that describes a function in terms of its value on smaller inputs.

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

- Recurrences arise when an algorithm contains recursive calls to itself
- What is the actual running time of the algorithm?
- Need to solve the recurrence
  - Find an explicit formula of the expression
  - Bound the recurrence by an expression that involves n

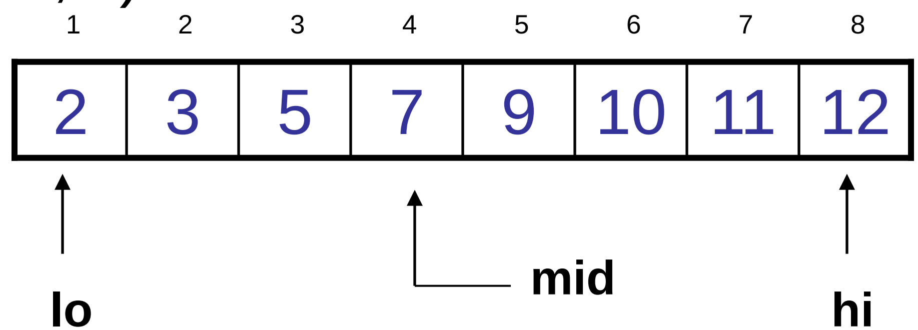
Example Recurrences

- $T(n) = T(n-1) + n$ $\Theta(n^2)$
  - Recursive algorithm that loops through the input to eliminate one item
- $T(n) = T(n/2) + c$ $\Theta(\lg n)$
  - Recursive algorithm that halves the input in one step
- $T(n) = T(n/2) + n$ $\Theta(n)$
  - Recursive algorithm that halves the input but must examine every item in the input
- $T(n) = 2T(n/2) + 1$ $\Theta(n)$
  - Recursive algorithm that splits the input into 2 halves and does a constant amount of other work

Recurrent Algorithms
BINARY-SEARCH

- for an ordered array A, finds if **x** is in the array A[lo…hi]

Alg.: BINARY-SEARCH (A, lo, hi, x)



if (lo > hi)

  return FALSE

mid ← $\lfloor(lo+hi)/2\rfloor$

if x = A[mid]

  return TRUE

if (x < A[mid])

  BINARY-SEARCH (A, lo, mid-1, x)

if (x > A[mid])

  BINARY-SEARCH (A, mid+1, hi, x)

Analysis of BINARY-SEARCH

Alg.: BINARY-SEARCH (A, lo, hi, x)

```
if (lo > hi)                                   ← constant time: c1
    return FALSE
mid ← ⌊(lo+hi)/2⌋                              ← constant time: c2
if x = A[mid]                                  ← constant time: c3
    return TRUE
if ( x < A[mid] )
    BINARY-SEARCH (A, lo, mid-1, x)            ← same problem of size n/2
if ( x > A[mid] )
    BINARY-SEARCH (A, mid+1, hi, x)            ← same problem of size n/2
```

- $T(n) = c + T(n/2)$
  - $T(n)$ – running time for an array of size n

The substitution method

1. Guess a solution

2. Use induction to prove that the solution works

Substitution method

- Guess a solution
  - $T(n) = O(g(n))$
  - Induction goal: apply the definition of the asymptotic notation
    - $T(n) \le d\, g(n)$, for some $d > 0$ and $n \ge n_0$
  - Induction hypothesis: $T(k) \le d\, g(k)$ for all $k < n$ (strong induction)
- Prove the induction goal
  - Use the **induction hypothesis** to find some values of the constants d and n_0 for which the **induction goal** holds

Example: Binary Search

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

- Guess: $T(n) = O(\lg n)$
  - Induction goal: $T(n) \leq d \lg n$, for some d and $n \geq n_0$
  - Induction hypothesis: $T(n/2) \leq d \lg(n/2)$
- Proof of induction goal:

$$T(n) = T(n/2) + c \leq d \lg(n/2) + c$$

$$= d \lg n - d + c \leq d \lg n$$

$$\text{if: } -d + c \leq 0,\ d \geq c$$

- Base case?

Example 2

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

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le c\, n^2$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n-1) \le c(n-1)^2$ for all $k < n$
- Proof of induction goal:

$$T(n) = T(n-1) + n \le c\,(n-1)^2 + n$$

$$= cn^2 - (2cn - c - n) \le cn^2$$

$$\text{if: } 2cn - c - n \ge 0 \Leftrightarrow c \ge n/(2n-1) \Leftrightarrow c \ge 1/(2 - 1/n)$$

  - For $n \ge 1 \Rightarrow 2 - 1/n \ge 1 \Rightarrow$ any $c \ge 1$ will work

Example 3

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

- Guess: $T(n) = O(n \lg n)$
  - Induction goal: $T(n) \le cn \lg n$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le cn/2 \lg(n/2)$
- Proof of induction goal:

$$T(n) = 2T(n/2) + n \le 2c\,(n/2)\lg(n/2) + n$$

$$= cn \lg n - cn + n \le cn \lg n$$

$$\text{if: } -cn + n \le 0 \Rightarrow c \ge 1$$

- Base case?

Changing variables

$$T(n) = 2T(\sqrt{n}\ ) + \lg n$$

– Rename: $m = \lg n \Rightarrow n = 2^m$

$T\ (2^m) = 2T(2^{m/2}) + m$

– Rename: $S(m) = T(2^m)$

$S(m) = 2S(m/2) + m \Rightarrow S(m) = O(m\lg m)$ (demonstrated before)

$T(n) = T(2^m) = S(m) = O(m\lg m)=O(\lg n\lg\lg n)$

Idea: transform the recurrence to one that you have seen before

The recursion-tree method

Convert the recurrence into a tree:

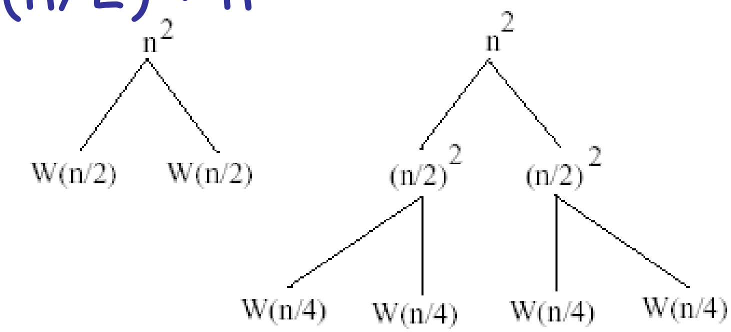
– Each node represents the cost incurred at various levels of recursion

– Sum up the costs of all levels

Used to “guess” a solution for the recurrence

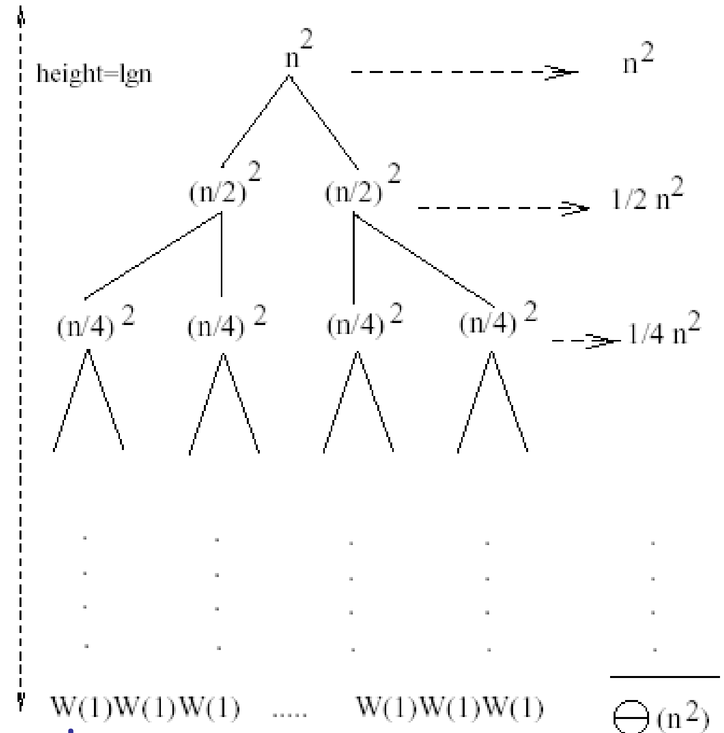
Example 1

$W(n) = 2W(n/2) + n^2$



$W(n/2) = 2W(n/4) + (n/2)^2$

$W(n/4) = 2W(n/8) + (n/4)^2$



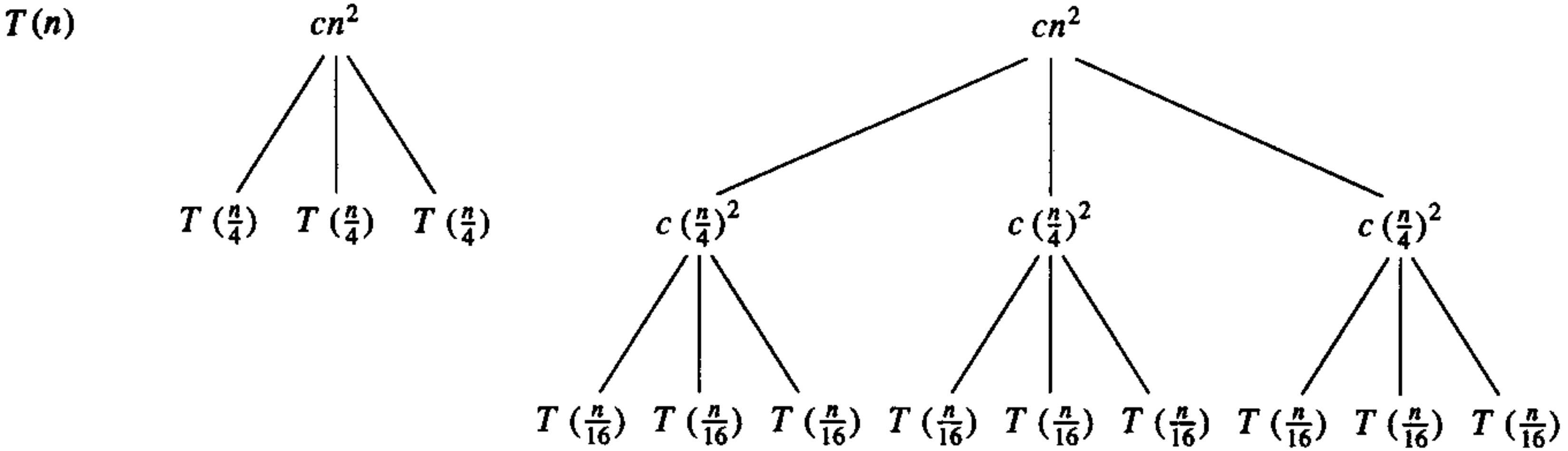
- Subproblem size at level i is: $n/2^i$
- Subproblem size hits 1 when $1 = n/2^i \Rightarrow i = \lg n$
- Cost of the problem at level i = $(n/2^i)^2$ No. of nodes at level $i = 2^i$
- Total cost:

$$W(n) = \sum_{i=0}^{\lg n-1} \frac{n^2}{2^i} + 2^{\lg n} W(1) = n^2 \sum_{i=0}^{\lg n-1} \left(\frac{1}{2}\right)^i + n \le n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + O(n) = n^2 \frac{1}{1-\frac{1}{2}} + O(n) = 2n^2$$

$\Rightarrow W(n) = O(n^2)$

Example 2

E.g.: $T(n) = 3T(n/4) + cn^2$



- Subproblem size at level i is: $n/4^i$
- Subproblem size hits 1 when $1 = n/4^i \Rightarrow i = \log_4 n$
- Cost of a node at level i = $c(n/4^i)^2$
- Number of nodes at level i = $3^i \Rightarrow$ last level has $3^{\log_4 n} = n^{\log_4 3}$ nodes
- Total cost:

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) \le \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{1}{1 - \frac{3}{16}} cn^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2)$$

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

Example 2 - Substitution

$$T(n) = 3T(n/4) + cn^2$$

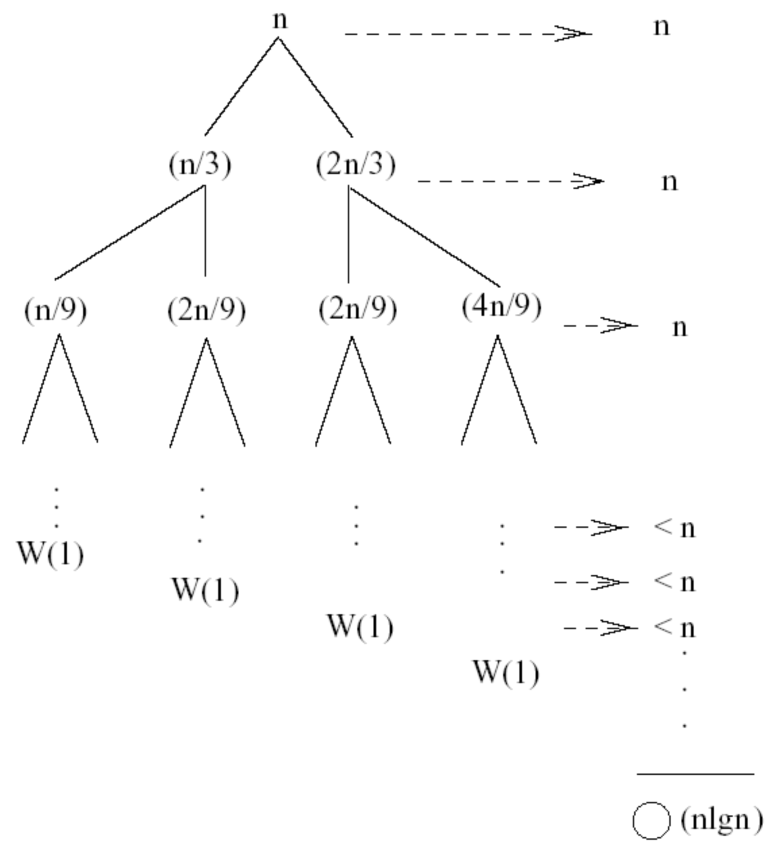
- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le dn^2$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/4) \le d\,(n/4)^2$
- Proof of induction goal:

$$\begin{aligned} T(n) &= 3T(n/4) + cn^2 \\ &\le 3d\,(n/4)^2 + cn^2 \\ &= (3/16)\,d\,n^2 + cn^2 \\ &\le d\,n^2 \qquad \text{if: } d \ge (16/13)c \end{aligned}$$

- Therefore: $T(n) = O(n^2)$

Example 3 (simpler proof)

$W(n) = W(n/3) + W(2n/3) + n$



- The longest path from the root to a leaf is:
  $n \to (2/3)n \to (2/3)^2\, n \to \ldots \to 1$
- Subproblem size hits 1 when
  $1 = (2/3)^i n \Leftrightarrow i = \log_{3/2} n$
- Cost of the problem at level i = n
- Total cost:

$$W(n) < n + n + \ldots = n(\log_{3/2} n) = n\frac{\lg n}{\lg \frac{3}{2}} = O(n \lg n)$$

$\Rightarrow W(n) = O(n\lg n)$

Example 3

$W(n) = W(n/3) + W(2n/3) + n$

- The longest path from the root to a leaf is:

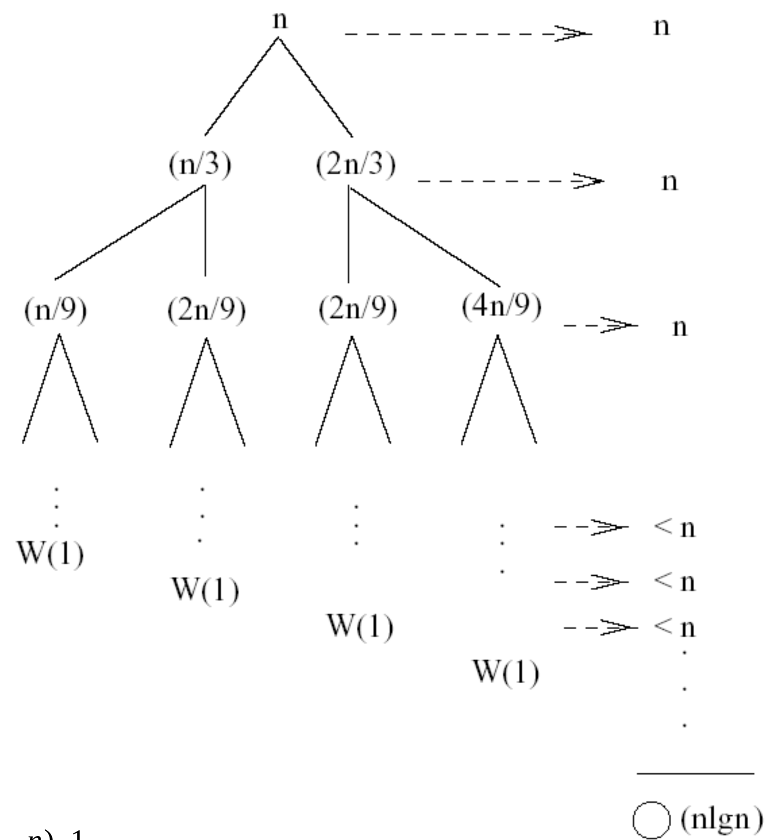
  $n \to (2/3)n \to (2/3)^2\, n \to \dots \to 1$

- Subproblem size hits 1 when

  $1 = (2/3)^i n \Leftrightarrow i = \log_{3/2} n$

- Cost of the problem at level i = n

- Total cost:



$$W(n) < n + n + \dots = \sum_{i=0}^{(\log_{3/2} n)-1} n + 2^{(\log_{3/2} n)} W(1) <$$

$$< n \sum_{i=0}^{\log_{3/2} n} 1 + n^{\log_{3/2} 2} = n \log_{3/2} n + O(n) = n \frac{\lg n}{\lg 3/2} + O(n) = \frac{1}{\lg 3/2} n \lg n + O(n)$$

$\Rightarrow W(n) = O(n\lg n)$

Example 3 - Substitution

$$W(n) = W(n/3) + W(2n/3) + O(n)$$

- Guess: $W(n) = O(n\lg n)$
  - Induction goal: $W(n) \le dn\lg n$, for some d and $n \ge n_0$
  - Induction hypothesis: $W(k) \le d\, k\lg k$ for any $K < n$ $(n/3, 2n/3)$
- Proof of induction goal:

  Try it out as an exercise!!

- $T(n) = O(n\lg n)$