Recursive Algorithms

To solve a problem of size n –

- split the size n problem into one or more smaller problems of the same kind
- recursively solve the smaller problems
- compute the solution for the size n problem from the solution of the smaller problems

Running Time for Recursive Algorithms

Let T(n) be the running time to solve a problem of size n.

Recursive algorithms tend to have one of two forms:

- split off b elements to create smaller problems
  - $T(n) = a\,T(n-b) + f(n)$ where $f(n) = 0$ or $\Theta(n^c \log^d n)$
- divide into subproblems of size n/b
  - $T(n) = a\,T(n/b) + f(n)$ where $\Theta(n^c \log^d n)$
  -
  - $a \geq 1$ is the number of smaller problems
  - f(n) is the work to split the size n problem into smaller problems and to combine the solutions to the smaller problems into the solution for the size n problem

Solving Recurrence Relations

$T(n) = a\,T(n-b) + f(n)$ where $f(n) = \Theta(n^c \log^d n)$

Cases are based on the number of subproblems and f(n).

a	f(n)	behavior	solution
> 1	any	base case dominates (too many leaves)	$T(n) = \Theta(a^{n/b})$
1	≥ 1	all levels are important	$T(n) = \Theta(n\,f(n))$

Solving Recurrence Relations

$T(n) = a\,T(n/b) + f(n)$ where $f(n) = \Theta(n^c \log^d n)$

Cases are based on the relationship between the number of subproblems, the problem size, and f(n).

(log a)/(log b) vs c	d	behavior	solution
<	any	top level dominates – more work splitting/combining than in subproblems (root too expensive)	$T(n) = \Theta(f(n))$
=	> -1	all levels are important – log n steps to get to base case, and roughly same amount of work in each level	$T(n) = \Theta(f(n) \log n)$
=	< -1	base cases dominate – so many subproblems that taking care of all the base cases is more work than splitting/combining (too many leaves)	$T(n) = \Theta(n^{(\log a)/(\log b)})$
>	any		