Analysis of Algorithms II

Johannes Borgström
johannes.borgstrom@it.uu.se

Review

Last lecture:

- Overview of Algorithm Analysis
- Recurrences
- From Code to Recurrences

This lecture:

- Solving Recurrences (finding closed forms)

Overview

Solution Techniques for Recurrences

1. Expansion method (plus proof by induction)
2. Substitution method (plus proof by induction)
3. Applying known theorem
4. Recursion tree method (plus proof by induction)

Note: No general way of solving recurrences.

Expansion Method

Expansion Method

Recall the `sumList` function:

```
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

Its runtime $T(n)$, where n is the length of the argument list, is given by the following recurrence:

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_1 & \text{if } n > 0 \end{cases}$$

A closed form is easier to work with, but how do we find one?

Expansion Method (cont.)

The **expansion method** aims to detect a pattern after evaluating the recurrence for several values.

If the result is similar (upon variable substitution) to one seen before, then we guess a similar closed form.

Consider

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_1 & \text{if } n > 0 \end{cases}$$

Expanding:

$$\begin{aligned} T(0) &= t_0 \\ T(1) &= T(0) + t_1 = 1 \cdot t_1 + t_0 \\ T(2) &= T(1) + t_1 = 2 \cdot t_1 + t_0 \\ T(3) &= T(2) + t_1 = 3 \cdot t_1 + t_0 \end{aligned}$$

Closed form (guess): $T(n) = n \cdot t_1 + t_0$

Proof by Induction

Let

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_1 & \text{if } n > 0 \end{cases}$$

Theorem: $T(n) = n \cdot t_1 + t_0$, for all $n \geq 0$.

Proof:

Base case: If $n = 0$, then $T(n) = t_0 = 0 \cdot t_1 + t_0$.

Inductive step: Assume $T(k) = k \cdot t_1 + t_0$ for some $k \geq 0$. Then

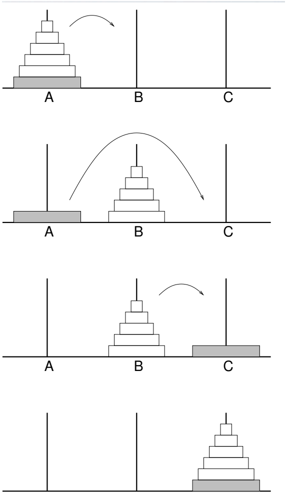
$$\begin{aligned} T(k+1) &= T(k) + t_1 \\ &= k \cdot t_1 + t_0 + t_1 \\ &= (k+1) \cdot t_1 + t_0. \end{aligned}$$

$\square$

Substitution Method

Example: Tower of Hanoi

1. Recursively move $n-1$ disks from tower A to B using C.
2. Move one disk from A to C
3. Recursively move $n-1$ disks from B to C using A.



Example: Tower of Hanoi

```
hanoi 0 from via to = ""
hanoi n from via to =
    hanoi (n-1) from to via ++
    from ++ "->" ++ to ++ " " ++
    hanoi (n-1) via from to
```

Let $M(n)$ be the **number** of moves that must be made for solving the problem of the Towers of Hanoi with n disks (using the above strategy).

From the program, we get the recurrence

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot M(n-1) + 1 & \text{if } n > 0 \end{cases}$$

Substitution Method

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot M(n-1) + 1 & \text{if } n > 0 \end{cases}$$

The substitution method starts with $M(n)$ and unfolds until a pattern emerges. At this point a generalisation step is used.

Substitution Method (cont.)

$$
\begin{aligned}
M(n) &= 2M(n-1)+1, \quad \text{by definition} \\
&= 2(2M(n-2)+1)+1, \quad \text{by definition} \\
&= 4M(n-2)+3 \\
&= 8M(n-3)+7, \quad \text{by definition and arithmetic} \\
&= 2^3 M(n-3)+(2^3-1), \quad \text{by arithmetic} \\
&= \cdots \\
&= 2^k M(n-k)+(2^k-1), \quad \text{by generalisation } 3 \rightsquigarrow k \\
&= \cdots \\
&= 2^n M(0)+(2^n-1), \quad \text{when } k=n \\
&= 2^n-1
\end{aligned}
$$

Proof by Induction

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot M(n-1) + 1 & \text{if } n > 0 \end{cases}$$

Theorem: $M(n) = 2^n - 1$, for **all** $n \geq 0$.

Proof:

Base case: If $n = 0$, then $M(n) = 0 = 2^0 - 1$.

Inductive step: Assume $M(k) = 2^k - 1$ for some $k \geq 0$. Then

$$\begin{aligned} M(k+1) &= 2 \cdot M(k) + 1, && \text{by definition} \\ &= 2(2^k - 1) + 1, && \text{by the induction hypothesis} \\ &= 2^{k+1} - 2 + 1 \\ &= 2^{k+1} - 1. \end{aligned}$$

$\square$

Tower of Hanoi

Hence the "move complexity" of `hanoi n ...` is $\Theta(2^n)$.

Note that $2^{64} - 1 \approx 1.8 \times 10^{19}$ moves, at 1 move/second, will take about 585 billion years.
The Big Bang is (currently) conjectured to have been only 15 billion years ago ...

Applying Known Theorem

Applying Known Theorem

"Doctor Theorem" (proof omitted): If $C(n) \geq 0$ for all n, and there exist **constants** $n_0 \geq 0$ and $a \geq 1$ such that

$$C(n) = a \cdot C(n-1) + \Theta(1)$$

for all $n > n_0$, then the closed form of the recurrence is

$$C(n) = \begin{cases} \Theta(n) & \text{if } a = 1 \\ \Theta(a^n) & \text{if } a > 1 \end{cases}$$

Applying Known Theorem

Consider the formula for the Tower of Hanoi

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2M(n-1) + 1 & \text{if } n > 0 \end{cases}$$

Applying the “Doctor Theorem” gives

$$M(n) = \Theta(2^n)$$

as before.

Master Theorem (A Taste)

We have already observed that recurrences of the form

- $T(n) = T(n-1) + \Theta(1)$ give $T(n) = \Theta(n)$, and
- $T(n) = T(n-1) + \Theta(n)$ give $T(n) = \Theta(n^2)$.

Divide-and-conquer algorithms lead to recurrences of the form

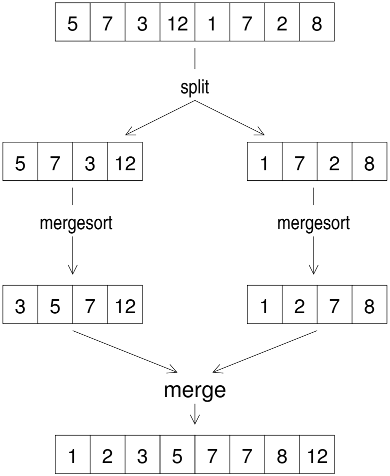
$$T(n) = aT(n/b) + f(n)$$

where a subproblems are produced, each of size n/b, and $f(n)$ is the time for **dividing** the input and **combining** the recursive results.

A pre-existing technique for solving such recurrences is the **Master Theorem** (no longer taught in PKD).

Recursion Tree Method

Practice: Merge Sort



Practice: Merge Sort: The Code

```haskell
split :: [a] -> ([a], [a])
split l = let t = (length l) `div` 2
          in (take t l, drop t l)

merge :: Ord a => [a] -> [a] -> [a]
merge [] m = m
merge l [] = l
merge l@(x:xs) m@(y:ys) =
  if x > y then y : merge l ys
  else x : merge xs m

sort :: Ord a => [a] -> [a]
sort [] = []
sort [x] = [x]
sort xs = let (ys, zs) = split xs
          in merge (sort ys) (sort zs)
```

Practice: Analysis of Split

```
split :: [a] -> ([a], [a])
split l = let t = (length l) `div` 2
          in (take t l, drop t l)
```

- `length l` takes time $\Theta(|\mathtt{l}|)$.
- ``a `div` b`` takes time $\Theta(1)$.
- `take t l` takes time $\Theta(\mathtt{t}) = \Theta(\lfloor \frac{|\mathtt{l}|}{2} \rfloor)$.
- `drop t l` takes time $\Theta(\mathtt{t}) = \Theta(\lfloor \frac{|\mathtt{l}|}{2} \rfloor)$.

Thus, `split l` always takes time $\Theta(|\mathtt{l}|) + \Theta(1) + 2 \cdot \Theta(\lfloor \frac{|\mathtt{l}|}{2} \rfloor) = \Theta(|\mathtt{l}|)$.

Exercise: Implement `split` using only one traversal of `l`.

Practice: Analysis of Merge

```
merge [] m = m
merge l [] = l
merge l@(x:xs) m@(y:ys) =
  if x > y then y : merge l ys
  else x : merge xs m
```

$$
\begin{aligned}
T_{\texttt{merge}}(0, |m|) &= \Theta(1) \\
T_{\texttt{merge}}(|l|, 0) &= \Theta(1) \\
T_{\texttt{merge}}(|l|, |m|) &= \max(\Theta(1) + T_{\texttt{merge}}(|l|, |m| - 1), \\
&\qquad \Theta(1) + T_{\texttt{merge}}(|l| - 1, |m|))
\end{aligned}
$$

max is used to reflect taking the branch of the if-expression that gives the *worst* execution time.

`merge l m` always takes $\Theta(|l| + |m|)$ time *at worst*.

Recursion Tree Method: Analysis of Sort

```
sort [] = []
sort [x] = [x]
sort xs = let (ys, zs) = split xs
          in merge (sort ys) (sort zs)
```

Let $T_{\texttt{sort}}(n)$ be the time of running sort on a list with n elements.

Recursion Tree Method: Analysis of Sort

Base cases $(n \leq 1)$:

Constructing a list of 0 or 1 element takes $\Theta(1)$ time.

Recursive case $(n > 1)$:

Divide: `split xs` takes $\Theta(|xs|) = \Theta(n)$ time.

Conquer: Recursive calls `sort ys` and `sort zs` each take time $T_{\texttt{sort}}(n/2)$ when n is even (since $|ys| = |zs| = n/2$ in this case), and $T_{\texttt{sort}}(\frac{n-1}{2}) / T_{\texttt{sort}}(\frac{n+1}{2})$ when n is odd (since $|ys| = |zs| - 1$ in this case).

Combine: `merge (sort ys) (sort zs)` takes $\Theta(n)$ time, since $|\texttt{sort}\ l| = |l|$ and thus $|\texttt{sort}\ ys| + |\texttt{sort}\ zs| = |ys| + |zs| = n$.

Recursion Tree Method: Analysis of Sort

Hence the runtime recurrence is:

$$T_{\texttt{sort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ \Theta(n) + 2T_{\texttt{sort}}(n/2) + \Theta(n) & \text{if } n > 1, n \text{ even} \\ \Theta(n) + T_{\texttt{sort}}(\frac{n-1}{2}) + T_{\texttt{sort}}(\frac{n+1}{2}) + \Theta(n) & \text{if } n > 1, n \text{ odd} \end{cases}$$

which simplifies to

$$T_{\texttt{sort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T_{\texttt{sort}}(n/2) + \Theta(n) & \text{if } n > 1, n \text{ even} \\ T_{\texttt{sort}}(\frac{n-1}{2}) + T_{\texttt{sort}}(\frac{n+1}{2}) + \Theta(n) & \text{if } n > 1, n \text{ odd} \end{cases}$$

where $\Theta(n)$ combines the total running time for dividing the input and combining the recursive results.

Now what?

Simplification

Instead of

$$T_{\texttt{sort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T_{\texttt{sort}}(n/2) + \Theta(n) & \text{if } n > 1, n \text{ even} \\ T_{\texttt{sort}}(\frac{n-1}{2}) + T_{\texttt{sort}}(\frac{n+1}{2}) + \Theta(n) & \text{if } n > 1, n \text{ odd} \end{cases}$$

work with

$$T_{\texttt{sort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T_{\texttt{sort}}(n/2) + \Theta(n) & \text{if } n = 2^k \text{ for some } k \geq 1 \end{cases}$$

Powers of 2 make the calculations easier—integers only.

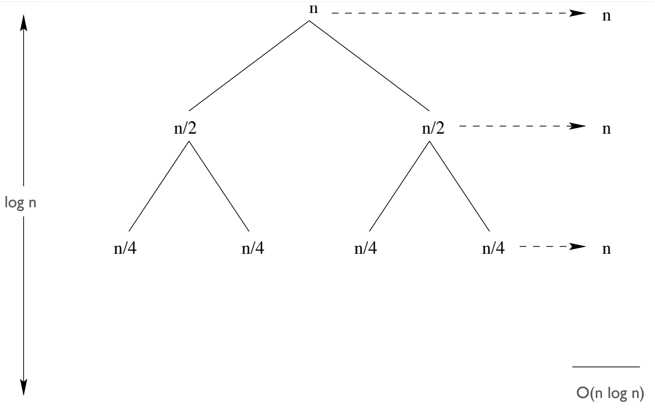
(For a complete analysis, we would also have to consider lists whose length is not a power of 2! The complexity of `sort` turns out to be the same on those.)

Recursion Tree Method

A **recursion tree** visualises the expansion of a recursion.

Can be used for **guessing** a closed form, not for proving it.

The recursion tree for the merge sort recurrence is:



Theorem: Runtime Complexity of Merge Sort

Theorem: If

$$T_{\texttt{sort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T_{\texttt{sort}}(n/2) + \Theta(n) & \text{if } n = 2^k \text{ for some } k \geq 1 \end{cases}$$

then $T(n) = \Theta(n \log n)$, for all $n = 2^k$ with $k \geq 1$.

Thus the (best/worst/average) time complexity of merge sort is $\Theta(n \log n)$.

Runtime Complexity of Merge Sort: Proof

Proof:

Either by induction (somewhat complicated, because of constant factors and the $\Theta(n)$ term)
...

or by applying the **Master Theorem**.

Summary

Solution Techniques for Recurrences

1. Expansion method (plus proof by induction)
2. Substitution method (plus proof by induction)
3. Applying known theorem
4. Recursion tree method (plus proof by induction)