Data structures: reflection

We’ve looked at . . .

- some classic **abstract datatypes** (lists, stacks, queues, sets, dictionaries)
- various **concrete implementations** of them (via extensible arrays, linked lists, hash tables, red-black trees . . . )

We’ve analysed their pros/cons in terms of asymptotic runtimes for operations. (Measured as number of line executions, paying attention to what’s allowed as a $\Theta(1)$ time **basic memory operation**.)

The above datatypes are used frequently in programming – and many other algorithms build on them.

Most of these data structures already provided in standard libraries (e.g. Java API classes).

But understanding of runtime characteristics can help in

- writing efficient **programs**
- constructing efficient **database queries**.
Recursion: a recurring theme

As we’ve seen, many algorithms can be presented as recursive: i.e. they involve subcall to (one or more instances of) same problem.

Examples:

- **Expmod**\((a,n,m)\) involves call to **Expmod**\((a,\lfloor n/2 \rfloor,m)\).
- **Mergesort**\((A,m,n)\) calls **Mergesort**\((A,m,p)\) and **Mergesort**\((A,p,n)\).
- **Insert**\((x,k)\) (for binary trees) may call **Insert**\((x.left,k)\) or **Insert**\((x.right,k)\).

Common pattern:

- ‘Simple’ (e.g. small) instances can be dealt with directly.
- For larger instances, may do work before/during/after the recursive call(s): we divide into subproblems, conquer these, combine results.

E.g. for Mergesort:

- **divide** is simply checking \(n - m > 1\) and computing \(\lfloor (m + n)/2 \rfloor\).
- **combine** is *merging* the two lists returned by the recursive calls.
Recurrence relations

How can we calculate the (asymptotic) runtime for a recursive algorithm?

E.g. write $T(n)$ for the worst-case runtime for Mergesort on array segments of size $n$.

```
MergeSort (A,m,n):
    if n−m = 1
        return [ A(m) ]
    else
        p = ⌊(m+n)/2⌋
        B = MergeSort (A,m,p)
        C = MergeSort (A,p,n)
        D = Merge (B,C)
        return D
```

Whatever the function $T$ is, it will satisfy

$$T(n) = T(⌊n/2⌋) + T(⌈n/2⌉) + F(n) \text{ for all } n > 1,$$

where $F(n)$ is the worst-case time for the divide and combine phases on inputs of size $n$. Can also say $T(1)$ is a constant $C$. 

*IADS Lecture 10 Slide 4*
Recurrence relations, continued

\[ T(n) = \begin{cases} 
  C & \text{if } n = 1 \\
  T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + F(n) & \text{otherwise}
\end{cases} \]

This is an example of a recurrence relation.
If we know \( C \) and \( F \), can compute \( T(n) \) for a specific \( n \), e.g.

\[ T(4) = 2T(2) + F(4) = 2(2T(1) + F(2)) + F(4) = 4C + 2F(2) + F(4) \]

But can we ‘solve’ the rec. rel. to find an explicit formula for \( T(n) \)?
Or at least, for its asymptotic growth rate?
Recurrence relations for growth rates

\[
T(n) = \begin{cases} 
  C & \text{if } n = 1 \\
  T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + F(n) & \text{otherwise}
\end{cases}
\]

Actually, if we only want the growth rate of \( T \), don’t need to know \( F \) precisely — knowing its growth rate is enough.

E.g. in Mergesort example, have \( F(n) = \Theta(n) \) (time for \textbf{Merge} on lists of length \( n/2 \)).

Leads to the concept of an asymptotic recurrence relation. E.g.

\[
T(n) = \begin{cases} 
  \Theta(1) & \text{if } n = 1 \\
  2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases}
\]

Solution we’re seeking isn’t a precise function, but a growth rate.

(\textit{Omission of } \lfloor - \rfloor \text{ and } \lceil - \rceil \text{ a bit sloppy . . . but can be shown these ‘don’t affect asymptotic solution’ in cases like this.})
Recurrence relations ctd.

Asymp. rec. relation for Mergesort again:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases}
\]

In Lecture 5 we saw informally that in this case \( T(n) = \Theta(n \lg n) \).

Other examples:

- Runtime of \texttt{Expmod}(a,n,m) for fixed a,m:
  \[
  T(n) = T(n/2) + \Theta(1) \quad \text{for } n > 1
  \]

- Runtime of \texttt{Exp}(a,n) for fixed a (\texttt{Expmod} without the \texttt{mod}):
  \[
  T(n) = T(n/2) + \Theta(n^2) \quad \text{for } n > 1
  \]

\* Can we solve such recurrences \textit{systematically}?
Is there a general pattern here?
How do we come up with solutions?

Approach 1: Use intuition/experience/numerical data to ‘guess’ a solution, then verify it using induction.

Usual concept of induction may need extending a bit.
E.g. for MergeSort:

Ordinary induction:

```
1 2 3 4 5 6
```

‘Log induction’:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

Note: log induction on array size $n \sim$ ordinary induction on MergeSort recursion depth.

IADS Lecture 10 Slide 8
The Master Theorem

**Approach 2:** If our recurrence just happens to be of the form . . .

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq n_0 \\
 aT(n/b) + \Theta(n^k) & \text{if } n > n_0 
\end{cases} \]

. . . then there’s a Master Theorem that simply gives us the answer. (Also works with ‘floors and ceilings’ around.)

The answer depends on how \( a \) compares with \( b^k \) (will explain!). Equivalently, how \( e = \log_b a \) compares with \( k \).

\[ T(n) = \begin{cases} 
\Theta(n^e) & \text{if } e > k \\
\Theta(n^k \log n) & \text{if } e = k \\
\Theta(n^k) & \text{if } e < k 
\end{cases} \]

This applies in many (not all) commonly arising situations. (CLRS 4.5 gives a more general version of the theorem.)
Master Theorem: informal intuition

Think about **total** work done by all **divide** / **combine** phases at each recursion level. Does this increase or decrease as we go down?

- Larger \(a\) (no. of subproblems) means more work as we descend.
- But larger \(b\) means each subproblem is smaller. If divide/combine work is \(F(n) = \Theta(n^k)\), then reducing problem size by factor \(b\) will reduce this work by \(b^k\).
- So break-even point is when \(a = b^k\). In this case, amount of work is ‘essentially the same’ for all levels.
A bit more mathematical detail for those interested . . .

- If $a < b^k$, then the most work is done at the top level. Thereafter, amount of work roughly decreases in geometric progression, by factor $r = a/b^k < 1$.
  
  So total work will be roughly top-level work ($\Theta(n^k)$) times $1 + r + r^2 + \cdots \leq 1/(1 - r)$ (constant). Still $\Theta(n^k)$.

- If $a > b^k$, work increases by $r = a/b^k > 1$ as we descend. Around $\log_b(n)$ levels. So bottom-level exceeds top-level by

  $$r^{\log_b(n)} = b^{\log_b(r) \cdot \log_b(n)} = b^{\log_b(n) \cdot \log_b(a/b^k) = n^{e-k}}$$

  So total work comes out as $\Theta(n^k) \cdot \Theta(n^{e-k}) = \Theta(n^e)$.

- If $a = b^k$, all levels are ‘essentially the same’. So work is roughly (top-level work $\times$ number of levels), i.e. $\Theta(n^k \lg n)$. 

IADS Lecture 10 Slide 11
Master Theorem in action

- Mergesort recurrence again:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases}
\]

Here \( a = 2, b = 2, k = 1 \). So \( e = \log_b a = 1 \) and \( e = k \).
So we’re in the middle case: \( \Theta(n \log n) \).

- \( \text{Exp}(a,n) \) for fixed \( a \):

\[
T(n) = T(n/2) + \Theta(n^2) \quad \text{if } n > 1
\]

Here \( a = 1, b = 2, k = 2 \). So \( e = \log_b a = 0 \) and \( e < k \).
Work at top-level dominates: solution is \( \Theta(n^2) \).

- Karatsuba algorithm for multiplying two \( n \)-digit numbers:

\[
T(n) = 3T(n/2) + \Theta(n) \quad \text{if } n > 1
\]

Here \( a = 3, b = 2, k = 1 \). So \( e = \log_2 3 \) and \( e > k \).
Solution is \( \Theta(n^{\log_2 3}) \) (cf. \( \Theta(n^2) \) for school method)
Thanks for listening!

Enjoy Aris’s lectures, and see you again in Sem 2 for some language processing and computability theory.

Reading for today’s lecture:
Roughgarden Chapter 4 (recommended)
CLRS Chapter 4, especially 4.5
KT Chapter 5 (many good examples, doesn’t explicitly state MT)
GTG 4.2 (relevant, again doesn’t explicitly state MT)