Asymptotic Algorithm Analysis & Sorting

(Version of 5th March 2010)

(Based on original slides by John Hamer and Yves Deville)

We can analyse an algorithm without needing to run it, and in so doing we can gain some understanding of its likely performance.

This analysis can be done at design time, before the program is written. Even if the analysis is approximate, problems may be detected.

The notation used in the analysis is helpful in documenting software libraries. It allows programs using such libraries to be analysed without requiring analysis of the library source code (which is often not available).

We will mostly analyse the runtime. The same principles apply to memory consumption. In general, we speak of algorithm complexity.

Runtime Equations

Consider the following function to sum the elements of an integer list:

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

The runtime, T, of this function depends on the argument list.
Realistically assuming that [pattern matching and] the + operation
take the same time, k_1, regardless of the two numbers being added,
we can see that only the length of the list, n, matters to T.
Actually, $T(n)$ is inductively defined by the following recursive equation:

$$T(n) = \begin{cases} k_0 & \text{if } n = 0 \\ T(n-1) + k_1 & \text{if } n > 0 \end{cases}$$

for constants k_0 (the time of [pattern matching and] returning 0) and k_1.

Solving Recurrences

The expression for $T(n)$ is called a recurrence. We can use it for computing runtimes (given suitable values of k_0 and k_1), but it is a difficult form to work with.

We want a closed form (that is, a non-recursive equation), if possible.

It turns out that the following is an equivalent definition of $T(n)$:

$$T(n) = k_0 + nk_1$$

Much simpler! But (a) how do we get there, and (b) can we prove it?

Deriving Closed Forms

There is no general way of solving recurrences. Recommended method: First guess the answer, and then prove it by induction.
Suggestions for making a good guess:

- If the recurrence is similar to one seen before (possibly upon variable substitution), then guess a similar closed form.
- Expansion Method: Detect a pattern for several values. Example:

$$\begin{aligned} T(0) &= k_0 \\ T(1) &= T(0) + k_1 = k_0 + k_1 \\ T(2) &= T(1) + k_1 = k_0 + 2k_1 \\ T(3) &= T(2) + k_1 = k_0 + 3k_1 \end{aligned}$$

- Iterative / Substitution Method: p. 16; Recursion Tree Method: p. 31.

Proof by Induction

Let:

$$T(n) = \begin{cases} k_0 & \text{if } n = 0 \\ T(n-1) + k_1 & \text{if } n > 0 \end{cases} \quad (1)$$

Theorem: $T(n) = k_0 + nk_1$, for all $n \geq 0$.

Proof:

Basis: If $n = 0$, then $T(n) = k_0 = k_0 + 0k_1$.

Inductive Step: Assume $T(n) = k_0 + nk_1$ for some $n \geq 0$. Then:

$$\begin{aligned} T(n+1) &= T(n) + k_1, \text{ by the recurrence } (1) \\ &= (k_0 + nk_1) + k_1, \text{ by the assumption above} \\ &= k_0 + (n+1)k_1, \text{ by arithmetic laws} \end{aligned}$$

Back to Algorithm Analysis

The equation $T(n) = k_0 + nk_1$ is a useful, but approximate, predictor of the actual runtime of `sumList`. Even if k_0 and k_1 were measured accurately, the actual runtime would vary with every change in the hardware or software environment.

Therefore, no actual values of k_0 or k_1 are of any interest!

The only interesting part of the equation is the term that involves n.
The runtime of `sumList` is (within constant factor k_1) proportional to the length, n, of the list. Calling `sumList` with a list twice as long will approximately double the runtime.

We say that $T(n)$ is $\Theta(n)$, which is pronounced "[big-]Theta of n".

The Θ Notation

The Θ notation is used to denote a set of functions that increase at the same rate (within some constant bound).

Formally, $\Theta(g(n))$ is the set of all functions $h(n)$ that are bounded below by $c_1 \cdot g(n) \geq 0$ and above by $c_2 \cdot g(n)$, for some constants $c_1 > 0$ and $c_2 > 0$, when n gets sufficiently large, i.e., at least some constant $n_0 > 0$.

The function $g(n)$ in $\Theta(g(n))$ is called a complexity function.

We write or say $h(n) = \Theta(g(n))$ when we mean $h(n) \in \Theta(g(n))$.

The Θ notation is used to give asymptotically tight bounds.

Terminology

Let $\lg x$ denote $\log_2 x$, let $k \geq 2$ be a constant, and let variable n denote the input size:

Function	Growth Rate	
1	constant	sub-linear
$\lg n$	logarithmic	
$\lg^2 n$	log-squared	
n	linear	polynomial
$n \lg n$		
n^2	quadratic	
n^3	cubic	
k^n	exponential	exponential
$n!$		super-exponential
n^n		

Example

Theorem: $10 + 5 \cdot n + n^2 = \Theta(n^2)$.

Proof: We need to choose constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that

$$0 \leq c_1 \cdot n^2 \leq 10 + 5 \cdot n + n^2 \leq c_2 \cdot n^2$$

for all $n \geq n_0$. Dividing by n^2 (assuming $n > 0$) gives

$$0 \leq c_1 \leq \frac{10}{n^2} + \frac{5}{n} + 1 \leq c_2$$

The "sandwiched" term, $\frac{10}{n^2} + \frac{5}{n} + 1$, gets smaller as n gets larger. It peaks at 16 for $n = 1$, so we can pick $n_0 = 1$. It drops to 6 for $n = 2$ and becomes very close to 1 for $n = 1000$. It never gets less than 1, so we can pick $c_1 = 1$. It never exceeds 16, so we can pick $c_2 = 16$.

Exercise: Prove that $5 \cdot n^3 + 7 \cdot n^2 - 3 \cdot n + 4 \neq \Theta(n^2)$.

Keep Complexity Functions Simple

While it is formally (and trivially) correct to say that $10 + 5n + n^2$ is $\Theta(10 + 5n + n^2)$, the whole purpose of the Θ notation is to work with simple expressions. Thus, we often do not expect any arbitrary factors or lower-order terms inside a complexity function.

Simplify complexity functions by:

- Setting all constant factors to 1.
- Dropping all lower-order terms.

Since $\log_b x = \frac{1}{\log_c b} \log_c x$, where $\frac{1}{\log_c b}$ is just a constant factor, we shall use $\lg x$ in complexity functions.

Variations on Θ: The O and Ω Notations

Variants of Θ include O ("big-Oh"), which drops the lower bound, and Ω ("big-Omega"), which drops the upper bound.

Any linear function $a + b \cdot n$ is in $O(n^2)$, $O(n^3)$, etc, but not in $\Theta(n^2)$, $\Theta(n^3)$, etc. Any quadratic function $a + b \cdot n + c \cdot n^2$ is in $\Omega(n)$.

O and Ω are used when the actual runtime varies, depending on the inputs. For example, we will see that insertion sort runs in quadratic or linear time, depending on how close to sorted the inputs are.

O is used to give an asymptotic upper bound on the running time of an algorithm, and Ω is used to give an asymptotic lower bound, but no claims are made how tight these bounds are. Θ is used to give tight bounds, namely when the upper and lower bounds are the same.

Example: Towers of Hanoi

The end of the world, according to a Buddhist legend (or not?) ...

Initial state: Tower A has n disks stacked by decreasing diameter.
Towers B and C are empty.

Rules:

Only move one disk at a time.

Only move the top-most disk of a tower.

Only move a disk onto a larger disk.

Objective and final state: Move all the disks from tower A to tower C, using tower B, without violating any rules.

Problem: Write a program that outputs a (minimal) sequence of moves to be made for reaching the final state from the initial state, without violating any rules.

Strategy

1. Move $n-1$ disks from A to B, using C.
2. Move 1 disk from A to C.
3. Move $n-1$ disks from B to C, using A.

Specification and SML Program

```
hanoi(n,from,via,to)
TYPE: int * string * string * string -> string
PRE:  n >= 0
POST: description of the moves to be made for transferring
      n disks from tower from to tower to, using tower via
EX:   ...
VARIANT: n
fun hanoi(0,from,via,to) = ""
  | hanoi(n,from,via,to) =
      hanoi(n-1,from,to,via) ^ from ^ "->" ^ to ^ " " ^
      hanoi(n-1,via,from,to)
```

Will the end of the world be provoked by `hanoi(64,"A","B","C")`, even on the fastest computer of the year 2020?!

Analysis

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

From the SML program, we get the recurrence:

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

How to solve this recurrence?

- Guess the closed form and prove it.
- Guessing: By expansion, iterative / substitution, recursion-tree method.
- Proving: By induction, or by application of a pre-established formula.

Iterative / Substitution Method

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

Proof by Induction

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

Proof: **Basis:** If $n = 0$, then $M(n) = 0 = 2^0 - 1$.

Inductive Step:

Assume the theorem holds for $n - 1$, for some $n > 0$. Then:

$$\begin{aligned} M(n) &= 2 \cdot M(n-1) + 1, \text{ by the recurrence } (2) \\ &= 2 \cdot (2^{n-1} - 1) + 1, \text{ by the assumption above} \\ &= 2^n - 1, \text{ by arithmetic laws} \end{aligned}$$

Hence: The complexity of `hanoi(`n`,...)` is $\Theta(2^n)$.

Note that $2^{64} - 1 \approx 18.5 \cdot 10^{18}$ moves will take 580 billion years at 1 move / second, but the Big Bang is (currently) conjectured to have been only 15 billion years ago ...

Application of a Pre-Established Formula

Theorem 1: (proof omitted) If, for some constants a and b:

$$C(n) = a \cdot C(n-1) + b \quad \text{if } n > 0$$

then the closed form of the recurrence is:

$$C(n) = a^n \cdot C(0) + b \cdot \sum_{0 \le i < n} a^i$$

Particular cases:

- $a = 1$: $C(n) = C(0) + b \cdot n = \Theta(n)$
- $a = 2$: $C(n) = 2^n \cdot C(0) + b \cdot (2^n - 1) = \Theta(2^n)$
- $a \ge 2$: $C(n) = a^n \cdot C(0) + b \cdot \frac{a^n - 1}{a - 1} = \Theta(a^n)$

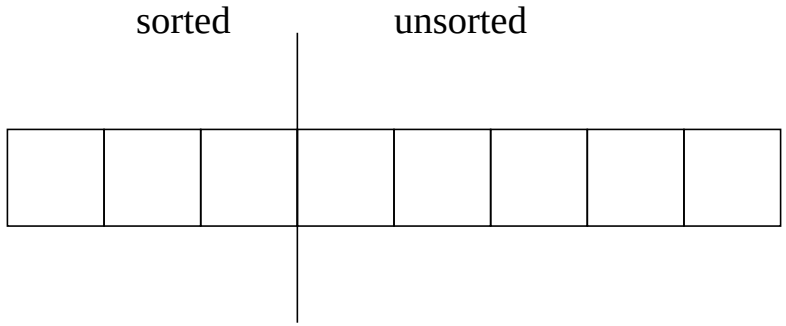
Another pre-established formula is in the Master Theorem (see below).

Sorting: Insertion Sort

Assume we must sort an array of n elements in non-decreasing order.

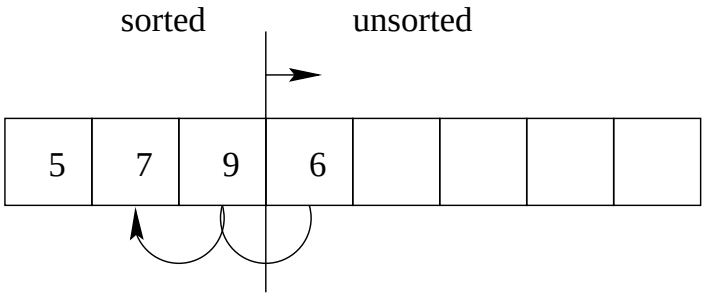
At any moment of insertion-sorting, it is divided into two sections:

- The sorted section (usually occupying the lower index positions).
- The section not looked at yet, often (and misleadingly) called the unsorted section.

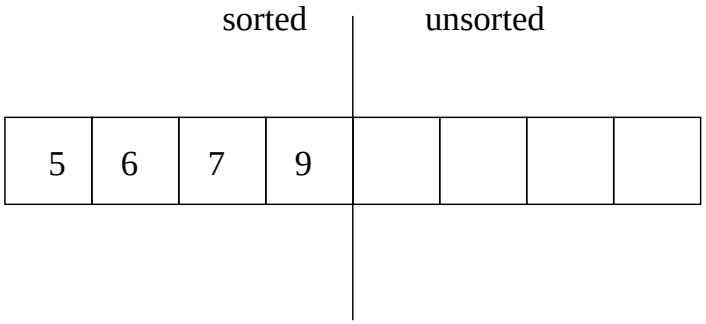


Place the dividing line between the first two elements (as a 1-element array is always sorted). While the line is not at the end of the array:
Advance the line one element to the right, swapping the newly encountered element with its predecessors while it is incorrectly located.

After the second step:



After the third step:



Analysis of Insertion Sort

Insertion sort runs in two loops.

- The outer loop, isort, processes each element in the array, inserting it into position.
- The inner loop, ins, inserts a single element into position.

The amount of work done by the inner loop depends on how many larger elements already exist in the sorted section of the array. If there are none, then the loop completes after a single comparison. If there are several, then the loop compares and swaps with each one.
In the worst case, every element is larger than the one being inserted.
The runtime for the inner loop, $T_{\text{ins}}(i)$, depends thus on the size i of the sorted section:

$$T_{\text{ins}}(i) = \Theta(i) \text{ at worst}$$

The runtime for the outer loop, $T_{\text{isort}}(n)$, (and hence the runtime for the whole sort) is the sum of these inner-loop runtimes:

$$T_{\text{isort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T_{\text{ins}}(1) + T_{\text{ins}}(2) + \cdots + T_{\text{ins}}(n-1) & \text{if } n > 1 \end{cases}$$

where n is the size of the array.

Equivalently, and as a recurrence:

$$T_{\text{isort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T_{\text{isort}}(n-1) + T_{\text{ins}}(n-1) & \text{if } n > 1 \end{cases} \tag{3}$$

If $T_{\text{ins}}(i) = T_{\leq} = k$ for **all** i (the best case: the array is initially already in sorted order), where k is a constant, then:

$$T_{\text{isort}}(n) = k(n-1) = \Theta(n)$$

If $T_{\text{ins}}(i) = (T_{\leq} + T_{\text{swap}})i = k'i$ for **all** i (the worst case: the array is initially in reverse-sorted order), where k' is a constant, then:

$$T_{\text{isort}}(n) = \sum_{j=1}^{n-1} k'j = k' \frac{n(n-1)}{2} = \Theta(n^2)$$

If $T_{\text{ins}}(i) = k' \frac{i}{2} = \frac{k'}{2} i$ on average (the average case), then (as in the worst case, since dividing a constant by 2 gives a constant):

$$T_{\text{isort}}(n) = \Theta(n^2)$$

Overall, we say that insertion-sort runs in $\Theta(n)$ time at best,
and in $\Theta(n^2)$ time on average and at worst.

Merge Sort (John von Neumann, 1945)

Runtime: $\Theta(n \lg n)$, where n is the number of elements to be sorted.

Apply the Divide & Conquer (& Combine) Principle:

```
function sort L
TYPE: int list -> int list
PRE:  (none)
POST: a non-decreasingly sorted permutation of L
EX:   sort [5,7,3,12,1,7,2,8,13] = [1,2,3,5,7,7,8,12,13]
VARIANT: |L|
fun sort [] = []
  | sort [x] = [x]   (* Question: Why indispensable?! *)
  | sort xs =
       let val (ys,zs) = split xs
       in  merge (sort ys) (sort zs) end
```

Splitting a List Into Two 'Halves'

```
split L
TYPE: 'a list -> ('a list * 'a list) ;  PRE: (none)
POST: (A,B) such that A@B is a permutation of L,
      but A and B are of the same length, up to one element
EX:   split [5,7,3,12,1,7,2,8,13] = ([5,7,3,12],[1,7,2,8,13])
```

Note that the order of the elements inside A and B is irrelevant!

Naïve SML program:

```
fun split L =
  let val t = (length L) div 2
  in  (List.take (L,t), List.drop (L,t)) end
```

Runtime: $n + \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{2} \rfloor = \Theta(2n) = \Theta(n)$, where $n = |\,\mathtt{L}\,|$.

Exercise: How to realise `split` in $\Theta(n)$ time with one traversal of `L`?

Merging Two Sorted Lists

```
merge L M
TYPE: int list -> int list -> int list
PRE:  L and M are non-decreasingly sorted
POST: a non-decreasingly sorted permutation of the list L@M
EX:   merge [3,5,7,12] [1,2,7,8,13] = [1,2,3,5,7,7,8,12,13]
VARIANT: |L|*|M|          (Exercise: Try |L|+|M| as variant.)
fun merge [] M = M
  | merge L [] = L
  | merge (L as x::xs) (M as y::ys) =
       if x > y then y :: merge L ys else x :: merge xs M
```

Runtime: $\Theta(|\texttt{L}| + |\texttt{M}|)$ at worst

Exercise: Redo all the functions in this lecture for polymorphic lists.

Analysis of Merge Sort

We can express the runtime of merge sort by the recurrence:

$$T_{\texttt{msort}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T_{\texttt{msort}}(n/2) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{4}$$

where $\Theta(1)$ is the time of the base case, and $\Theta(n)$ is the total time of dividing the list by `split` and combining the sub-results by `merge`, for a list of n elements.

The closed form is $T_{\texttt{msort}}(n) = \Theta(n \lg n)$, in **all** cases.

Merge sort is better than insertion sort in the worst and average cases, but insertion sort has the edge with nearly-sorted data.

Exercise: Redo the analysis for merge-sorting an array instead of a list.

Solving More Recurrences

We have already observed that a recurrence of the form

- $T(n) = T(n-1) + \Theta(1)$ gives a linear runtime (`sumList`, `ins`).
- $T(n) = T(n-1) + \Theta(n)$ gives a quadratic runtime (`isort`).

Divide-and-conquer algorithms produce recurrences of the form

$$T(n) = a \cdot T(n/b) + f(n) \tag{5}$$

when a sub-problems are produced, each of size n/b, and $f(n)$ is the total time of dividing the input and combining the sub-results.

There is a pre-established formula for looking up the closed forms of many recurrences of this form, based on the so-called master theorem.
We will first look at the special case for merge sort, and then study the general solution.

Proof of the Merge-Sort Recurrence

This proof gives the general flavour of solving divide-and-conquer recurrences. The formal proof is complicated by certain technical details, such as when n is not an integer power of b. We will ignore such issues in this proof, appealing instead to the intuition that runtime will increase in a more or less smooth fashion for intermediate values of n.

Theorem: If (compare with the recurrence (4) two pages ago)

$$T(n) = \begin{cases} 2 & \text{if } n = 2 = 2^k \text{ for } k = 1 \\ 2 \cdot T(n/2) + \Theta(n) & \text{if } n = 2^k \text{ for some } k > 1 \end{cases} \tag{6}$$

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

Proof by Induction

Proof: For $n = 2^k$ with $k \geq 1$, the closed form $T(n) = \Theta(n \lg n)$ becomes $T(2^k) = 2^k \lg 2^k$.

Basis: If $k = 1$ (and hence $n = 2$), then $T(n) = 2 = 2 \lg 2$.

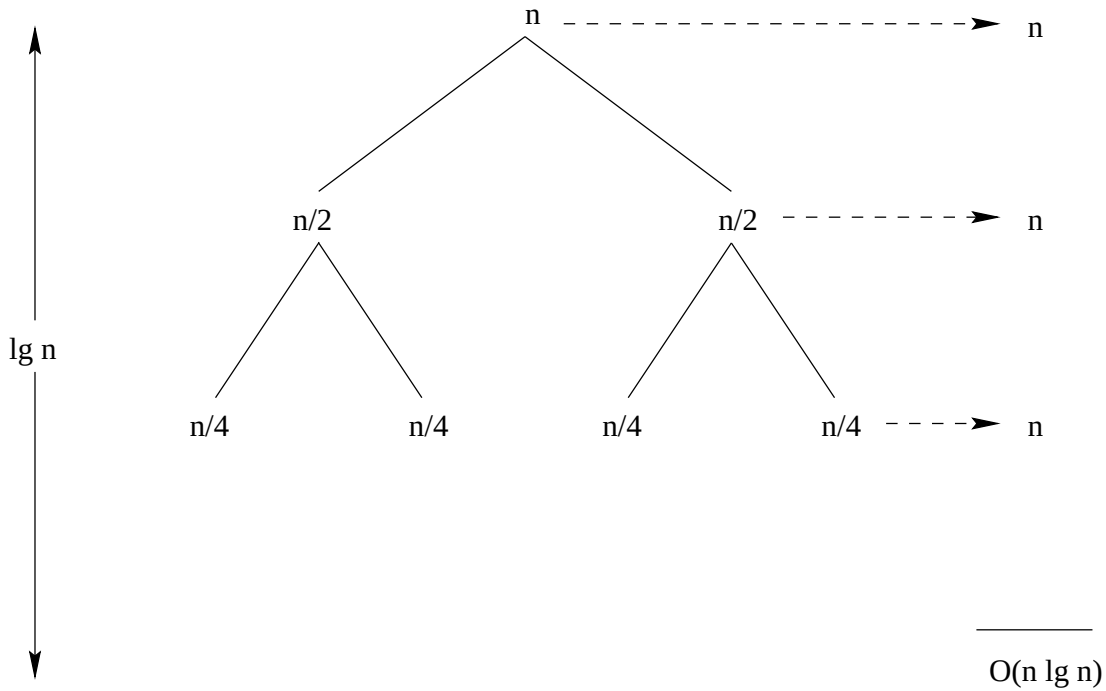
Inductive Step: Assume the theorem holds for some $k \geq 1$. Then:

$$\begin{aligned}
T(2^{k+1}) &= 2T(2^{k+1}/2) + 2^{k+1}, \text{ by the recurrence (6)} \\
&= 2T(2^k) + 2^{k+1}, \text{ by arithmetic laws} \\
&= 2(2^k \lg 2^k) + 2^{k+1}, \text{ by the assumption} \\
&= 2^{k+1}(\lg 2^k + 1), \text{ by arithmetic laws} \\
&= 2^{k+1}(\lg 2^k + \lg 2^1), \text{ by arithmetic laws} \\
&= 2^{k+1} \lg 2^{k+1}, \text{ by arithmetic laws}
\end{aligned}$$

The Recursion-Tree Method

Recursion trees visualise what happens when a recursion is expanded.

The recursion tree for the merge-sort recurrence looks like this:



The Master Method and Master Theorem

From now on, we will ignore the base cases of a recurrence.

The solution to a recurrence of the form $T(n) = a \cdot T(n/b) + f(n)$ reflects the "battle" between the two terms in the sum. Think of $a \cdot T(n/b)$ as the process of "distributing the work out" to $f(n)$, where the actual work is done.

Theorem 2: (also known as Master Theorem, proof omitted)

1. If $f(n)$ is dominated by $n^{\log_b a}$ (see the next page), then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n)$ and $n^{\log_b a}$ are balanced (that is, when $f(n) = \Theta(n^{log_b a})$), then $T(n) = \Theta(n^{log_b a} \lg n)$.
3. If $f(n)$ dominates $n^{\log_b a}$ and if the regularity condition (see next page) holds, then $T(n) = \Theta(f(n))$.

Dominance and the Regularity Condition

The three cases of the master theorem depend on comparing $f(n)$ to $n^{\log_b a}$. However, it is not sufficient for $f(n)$ to be "just a bit" smaller or bigger than $n^{\log_b a}$. Cases 1 and 3 only apply when there is a polynomial difference between the functions, that is when the ratio between the dominator and the dominee is asymptotically larger than the polynomial n^ϵ for some constant $\epsilon > 0$.

Example: n^2 is polynomially larger than both $n^{1.5}$ and $\lg n$.

Counter-Example: $n \lg n$ is not polynomially larger than n.

In Case 3, the regularity condition requires that $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n.
(All the f functions in this course will satisfy this condition.)

Gaps in the Master Theorem

The master theorem does not cover all possible recurrences of the form (5):

- The recurrence might not have the proper form.
- Cases 1 and 3: The difference between $f(n)$ and $n^{log_b a}$ might not be polynomial.

  Counter-Example: The master theorem does not apply to the recurrence $T(n) = 2 \cdot T(n/2) + n \lg n$, despite it having the proper form. We have $a = 2 = b$, so we need to compare $f(n) = n \lg n$ to $n^{\log_b a} = n^1 = n$. Clearly, $f(n) = n \lg n > n$, but the ratio $f(n)/n$ is $\lg n$, which is asymptotically less than the polynomial n^ϵ for any constant $\epsilon > 0$, and we are not in Case 3.
- Case 3: The regularity condition might not hold.

Common Special Cases of the Master Theorem

a	b	$n^{\log_b a}$	$f(n)$	Case	$T(n)$
1	2	n^0	$\Theta(1)$	2	$\Theta(\lg n)$
			$\Theta(\lg n)$	none	$\Theta(?)$
			$\Theta(n \lg n)$	3	$\Theta(n \lg n)$
			$\Theta(n^k)$, with $k > 0$	3	$\Theta(n^k)$
2	2	n^1	$\Theta(1)$	1	$\Theta(n)$
			$\Theta(\lg n)$	1	$\Theta(n)$
			$\Theta(n)$	2	$\Theta(n \lg n)$
			$\Theta(n \lg n)$	none	$\Theta(?)$
			$\Theta(n^k)$, with $k > 1$	3	$\Theta(n^k)$

Quicksort (Sir C. Antony R. Hoare, 1960)

Average-case runtime: $\Theta(n \lg n)$ for n elements.

Application of the Divide & Conquer (& Combine) Principle:

```
...  (* same specification as for merge sort! *)
VARIANT: |L|
fun sort [] = []   (* Question: Why no [x] base case?! *)
  | sort (x::xs) =
       let val (S,B) = partition(x,xs)
       in  (sort S) @ (x :: (sort B)) end
```

Double recursion and no tail-recursion.

Usage of $X@Y$ (concatenation), which takes $\Theta(|X|)$ time.

Partitioning a List

```
partition(p,L)                    (* p is called the 'pivot' *)
TYPE: int * int list -> int list * int list
PRE:  (none)
POST: (S,B) where S has all x<p of L and B has all x>=p of L
EX: partition(5,[7,3,12,1,7,2,8,13]) = ([3,1,2], [7,12,7,8,13])
VARIANT: |L|
fun partition(p,[]) = ([],[])
  | partition(p,x::xs) =
      let val (S,B) = partition(p,xs)
      in  if x < p then (x::S,B) else (S,x::B) end
```

Runtime: $\Theta(|L|)$

Generalisation by Accumulator Introduction

Average-case runtime: $\Theta(n \lg n)$, for $n = |\mathtt{L}|$, but much less memory.

```
sort' L A
TYPE: int list -> int list -> int list
PRE:  (none)
POST: (a non-decreasingly sorted permutation of L) @ A
EX:   sort' [5,7,3,12] [1,7,2,8,13] = [3,5,7,12,1,7,2,8,13]
VARIANT: |L|
local fun sort' [] A = A
        | sort' (x::xs) A =
            let val (S,B) = partition (x,xs)
            in  sort' S (x :: (sort' B A)) end
in fun sort L = sort' L [] end
```

Double recursion, but one tail-recursion. No concatenation (@).

Worst-Case Analysis of Quicksort

The seen simple version of quicksort always chooses the leftmost element as the pivot. If the list is already sorted, then the partition will always produce a left sub-list of size 0, and a right sub-list of size $n-1$.
The runtime recurrence for n elements is then

$$T_{\text{qsort}}(n) = \begin{cases} \Theta(1) & \text{if } n = 0 \\ T_{\text{qsort}}(n-1) + \Theta(n) & \text{if } n > 0 \end{cases}$$

like (3) for insertion sort,
and hence quicksort takes $\Theta(n^2)$ time in the worst case.

Best-Case and Average-Case Analysis of Quicksort

If partitioning always produces two equal-size lists,
then the runtime recurrence becomes

$$T_{\text{qsort}}(n) = \begin{cases} \Theta(1) & \text{if } n = 0 \\ 2 \cdot T_{\text{qsort}}(n/2) + \Theta(n) & \text{if } n > 0 \end{cases}$$

like (4) for merge sort, and hence (by Case 2 of the Master Theorem) quicksort takes $\Theta(n \lg n)$ time in the best case.

If partitioning always produces 9-to-1 splits (which is a poor average behaviour), then the recurrence becomes

$$T_{\text{qsort}}(n) = T_{\text{qsort}}(9n/10) + T_{\text{qsort}}(n/10) + \Theta(n)$$

This gives a recursion tree ending at depth $\log_{10/9} n = \Theta(\lg n)$
and hence quicksort takes $\Theta(n \lg n)$ time in the average case.

Stable Sorting

In practice, the data to be sorted is comprised of two parts: a key, which is used for comparison, and some additional satellite data. Two different data elements may thus compare "equal".

Example: When sorting a list of email messages by the day sent, all email messages sent on a given day will compare "equal".

A stable sort is one that never exchanges data elements with equal keys.

Examples: Insertion sort (as described above), merge sort (as with the merging and the naïve splitting above), and quicksort (as with the partitioning and the naïve pivot choice above) are stable.

Counter-Examples: Quicksort is not stable under most practical pivot choices and partitioning algorithms (used for arrays). Merge-sort is not stable when replacing `x>y` by `x>=y` in the `merge` function on page 26.