Topic 5: Algorithm Analysis & Sorting[1]
(Version of October 21, 2012)

Pierre Flener

Computing Science Division
Department of Information Technology
Uppsala University
Sweden

Course 1DL201:
Program Construction and Data Structures

[1]Based on original slides by John Hamer and Yves Deville, with some figures from the CLRS textbook (which are © The MIT Press, 2009)

Outline

Outline

Road Map

Data structures may offer at least the following operations:

- empty: Create an instance with zero elements.
- isEmpty: Return true if and only if there are 0 elements.
- insert: Insert an element with a given key and value.
- delete: Delete an element (if any) with a given key.
- search: Return a value (if any) for a given key.
- minimum: Return an element (if any) with the min key.
- maximum: Return an element (if any) with the max key.
- merge: Return the union of two instances.
- walk: List all the elements in a given order.
- sort: List all the elements in a given order on the keys.
- ...

Some data structures may have further specific operations or may not need to implement all of the operations above.

Outline

Asymptotic Algorithm Analysis

We can analyse an algorithm without needing to run it, and thus 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, performance 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 performance.
The same principles apply to memory consumption.
We speak of time complexity and space complexity.

Runtime Equations

Consider the following function, which returns the sum of the elements of the given integer list:

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

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

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

where t_0 (the time of [pattern matching and] returning 0) and t_{add} are constants (that is, they do not depend on n).

Solving Recurrences

The expression for $T(n)$ is called a recurrence.
We can use it for computing runtimes (given actual values of the constants t_0 and t_{add}), but it is difficult to work with.

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

Equivalent definition of $T(n)$, for all $n \geq 0$:

$$T(n) = n \cdot t_{\text{add}} + t_0$$

Much simpler! But: How do we get there? 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 (upon variable substitution) to one seen before, then guess a similar closed form.
- Expansion Method: Detect a pattern for several values. Example:

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

- Iterative / Substitution Method: page 22
- Recursion Tree Method: page 39

UPPSALA UNIVERSITET

Proof by Induction

Let:

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

Theorem: $T(n) = n \cdot t_{\text{add}} + t_0$, for all $n \geq 0$.

Proof

Basis: If $n = 0$, then $T(n) = t_0 = 0 \cdot t_{\text{add}} + t_0$, by (1).

Induction: Assume $T(n) = n \cdot t_{\text{add}} + t_0$ for some $n \geq 0$. Then:

$$\begin{aligned} T(n+1) &= T(n) + t_{\text{add}}, \text{ by the recurrence (1)} \\ &= (n \cdot t_{\text{add}} + t_0) + t_{\text{add}}, \text{ by the assumption above} \\ &= (n+1) \cdot t_{\text{add}} + t_0, \text{ by arithmetic laws} \quad \square \end{aligned}$$

Back to Algorithm Analysis

Road Map

Asymptotic Algorithm Analysis

Insertion Sort

Merge Sort

Master Method

Quick Sort

Accumulator Introduction

Stable Sorting

Road Map Revisited

The equation $T(n) = n \cdot t_{\text{add}} + t_0$ is a useful, but approximate, predictor of the actual runtime of `sumList`. Even if t_0 and t_{add} were measured accurately, the actual runtime would vary with every change in the hardware or software environment.

Therefore, no actual values of t_0 or t_{add} are of any interest!

The only interesting part of the equation is the term with n. The runtime of `sumList` is (within constant factor t_{add}) 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 $\Theta(n)$, pronounced "[big-]Theta of n".

UPPSALA UNIVERSITET

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 $f(n)$ 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, that is, when n is at least some constant $n_0 > 0$.

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

We write $f(n) = \Theta(g(n))$ when we mean $f(n) \in \Theta(g(n))$.

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

☞ From now on (except in induction proofs), we use $\Theta(1)$ instead of introducing constants such as t_0 and t_{add}.

Example

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

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

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

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

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

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

Exercise: Prove that $t_{add} \cdot n + t_0 = \Theta(n)$.

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

We can simplify complexity functions by:

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

Since $\log_b x = \frac{1}{\log_c b} \cdot \log_c x$, where $\frac{1}{\log_c b}$ is a constant factor (when the bases b and c are constants), we shall use $\lg x$ in complexity functions.

Time Complexity of Built-In Functions

Let $|D|$ denote the number of elements in data structure D (that is the length of D if D is a list):

Built-in Function	Time Complexity
pattern matching	$\Theta(1)$ time always
a `{+,-,*,/,div}` b	$\Theta(1)$ time always *
`Int.{max,min}` (a,b)	$\Theta(1)$ time always *
a `{<=,<,=,<>,>,>=}` b	$\Theta(1)$ time always *
h`::`T	$\Theta(1)$ time always
L `@` R	$\Theta(\lvert L\rvert)$ time always
`length` L	$\Theta(\lvert L\rvert)$ time always
`rev` L	$\Theta(\lvert L\rvert)$ time always
$S_1 \wedge S_2$	$\Theta(\lvert S_1\rvert + \lvert S_2\rvert)$ time always

* Simplifying practical assumption, not valid for Poly/ML

UPPSALA UNIVERSITET

Road Map

Asymptotic Algorithm Analysis

Insertion Sort

Merge Sort

Master Method

Quick Sort

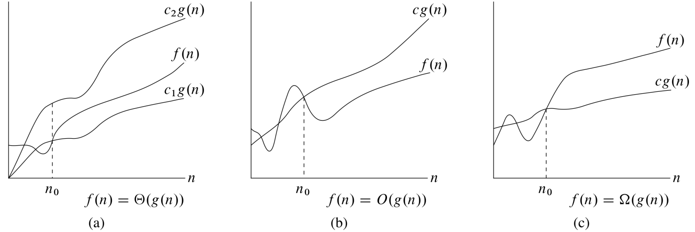
Accumulator Introduction

Stable Sorting

Road Map Revisited

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:

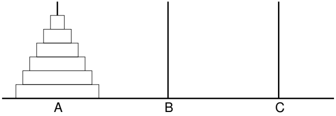


Examples: Any linear function $a \cdot n + b$ is in $O(n^2)$, $O(n^3)$, and so on, but not in $\Theta(n^2)$, $\Theta(n^3)$, and so on. Any quadratic function $a \cdot n^2 + b \cdot n + c$ is in $\Omega(n)$. We use O to give an upper bound on a function, and Ω to give a lower bound, but no claims are made about how tight these bounds are. We use Θ to give a tight bound, namely when the upper and lower bounds are the same.

Example: Towers of Hanoi

The end of the world, according to a Buddhist legend?

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



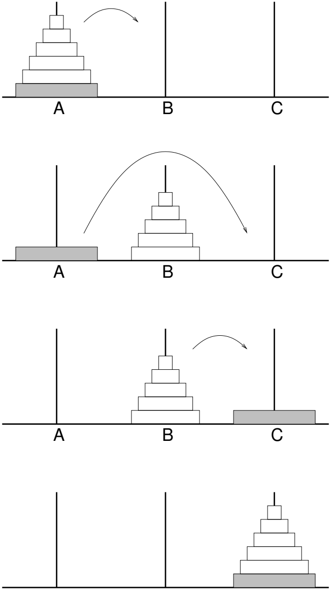
Rules

– Only move one disk at a time.
– Only move the top-most disk of a tower.
– Only move a disk onto a larger disk (if any).

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

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

Hanoi: Strategy



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

Hanoi: Specification and 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

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 the call

```
hanoi (64, "A", "B", "C")
```

even on the fastest computer of 20 years from now?!

Hanoi: 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 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 method, iterative / substitution method, or recursion-tree method.
- **Proving:** By induction, or by application of a pre-established formula.

UPPSALA UNIVERSITET

Road Map

Asymptotic Algorithm Analysis

Insertion Sort

Merge Sort

Master Method

Quick Sort

Accumulator Introduction

Stable Sorting

Road Map Revisited

Hanoi: Iterative / Substitution Method

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

Hanoi: Proof by Induction

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

Proof (note the difference with the approach on page 10!)

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

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

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

Hence: The move 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:

$$C(n) = \begin{cases} \Theta(1) & \text{if } n \leq b \\ a \cdot C(n-1) + \Theta(1) & \text{if } n > b \end{cases}$$

then the closed form of the recurrence is:

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

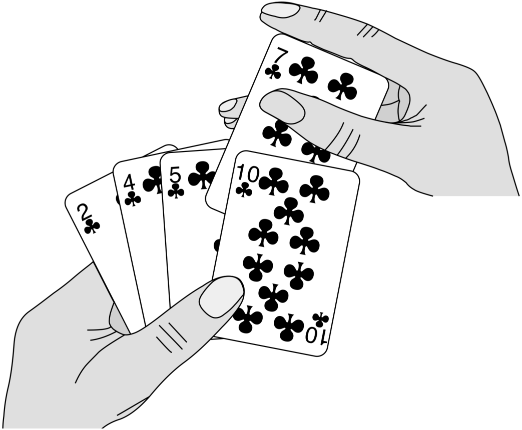
Another pre-established formula is in the Master Theorem:

page 44

Outline

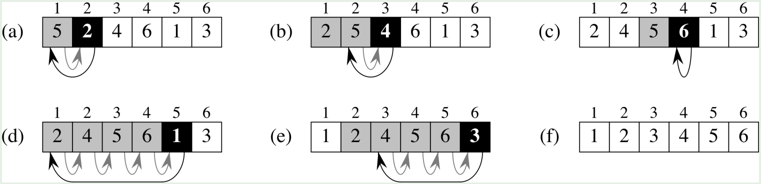
Insertion Sort

Assume we want to sort an array (a vector where elements can be changed) of *n* elements by non-decreasing order. The idea of the insertion sort algorithm is the same as many people use when sorting a hand of playing cards:



Example and Invariant

Example



At any moment of insertion sorting, the array is divided into:

- The sorted section (here at the lower indices).
- The section not looked at yet.

Example

Before step (b) above:

2	5	4	6	1	3

Algorithm

Initially place the dividing line between the first two elements (as a one-element array is always sorted).

While the dividing line is not after the last element: Advance the dividing line one notch to the right, and insert the newly encountered element into the sorted section.

Example

Before:

1	4	5 ‖	3	6	2

After:

1	3	4	5 ‖	6	2

Analysis

Insertion sort is implemented by two functions:

- The main function, called `sort`, processes each element, inserting it into the sorted section.
- The help function, called `ins`, is called by `sort` to insert one element into the sorted section.

The amount of work done by `ins` depends on how many larger elements are in the sorted section:

- If none, then a single comparison is performed.
- If several, then each of them is compared and moved.
- At worst, every element is larger than the inserted one.

The runtime for the help function `ins`, denoted by T_{ins}, depends thus on the size i of the sorted section:

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

Analysis

The runtime for the main function `sort`, denoted by T_{isort}, is the sum of the runtimes of the help function `ins`:

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

where n is the number of elements.

Equivalently, and as a recurrence:

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

Analysis

If $T_{\text{ins}}(i) = T_{\leq} = \Theta(1)$ for **all** i (the best case: the elements are initially already in sorted order), then:

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

If $T_{\text{ins}}(i) = i \cdot (T_{\leq} + T_{\text{move}}) = i \cdot \Theta(1)$ for **all** i (the worst case: the elements are initially in reverse-sorted order), then:

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

If $T_{\text{ins}}(i) = \frac{i}{2} \cdot (T_{\leq} + T_{\text{move}}) = i \cdot \Theta(1)$ on **average** for all i (the average case: the elements are moved on average half-way into the sorted section), then:

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

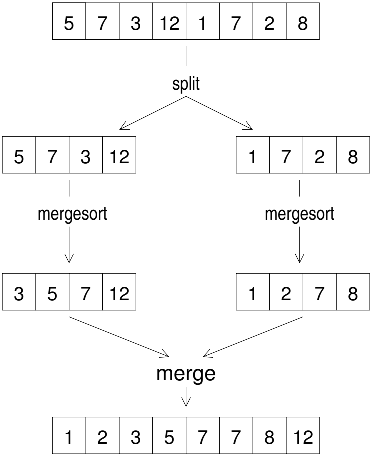
Overall, we say that insertion sort runs in $\Theta(n)$ time at best, and in $\Theta(n^2)$ time on average and at worst.

Outline

UPPSALA UNIVERSITET

Merge Sort (John von Neumann, 1945)

Runtime: Always $\Theta(n \cdot \lg n)$ for n elements.
Apply the divide & conquer (& combine) principle:



Road Map
Asymptotic Algorithm Analysis
Insertion Sort
Merge Sort
Master Method
Quick Sort
Accumulator Introduction
Stable Sorting
Road Map Revisited

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
EXAMPLE: split [5,7,3,12,1,7,2,8,13] = ([5,7,3,12],[1,7,2,8,13])
```

The order of the elements inside `A` and `B` is irrelevant!

Naïve program:

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

Exercise: `split` L always takes $|L| + 2 \cdot \left\lfloor \frac{|L|}{2} \right\rfloor = \Theta(|L|)$ time.

Exercise: How to realise `split` L 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
EXAMPLE: 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  (* Question: Why no other base cases? *)
  | merge (L as x::xs, M as y::ys) =
    if x > y then
        y :: merge (L, ys)
    else
        x :: merge (xs, M)
```

Question: `POST` suggests `fun merge(L,M) = sort(L@M)` :
Is that a good idea?

Exercise: `merge` (L, M) takes $\Theta(|L| + |M|)$ time at worst.

Merge Sort Program

```
sort L                    (* Question: Why not "mergesort L"? *)
TYPE: int list -> int list
PRE:  (none)
POST: a non-decreasingly sorted permutation of L
EXAMPLE: sort [5,7,3,12,1,7,2,8,13] = [1,2,3,5,7,7,8,12,13]

ALGORITHM: merge sort
VARIANT: |L|
TIME COMPLEXITY: Theta(|L| · lg |L|) always
fun sort [] = []
  | sort [x] = [x]        (* Question: Why indispensable?! *)
  | sort xs =
    let
        val (ys, zs) = split xs
    in
        merge (sort ys, sort zs)
    end
```

Analysis

Let $T_{\texttt{msort}}(n)$ be the time of running `sort` on n elements:

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, by page 34.
- **Conquer:** The recursive calls `sort` ys and `sort` zs take $T_{\texttt{msort}}\left(\frac{n}{2}\right)$ time each, because $|ys| + |zs| = n$ and $||ys| - |zs|| \leq 1$, by the post-condition of `split`. (If n is odd, then $\frac{n}{2}$ is not an integer, but this does not matter asymptotically.)
- **Combine:** `merge(sort` ys`, sort` zs`)` takes $\Theta(n)$ time, by page 35, since $|\texttt{sort}\ L| = |L|$ (by the post-condition of `sort`) and thus $|\texttt{sort}\ ys| + |\texttt{sort}\ zs| = |ys| + |zs| = n$ (by the post-condition of `split`).

Analysis (continued)

Hence the runtime recurrence:

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

which simplifies into:

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

where $\Theta(n)$ is the total time of dividing and combining.

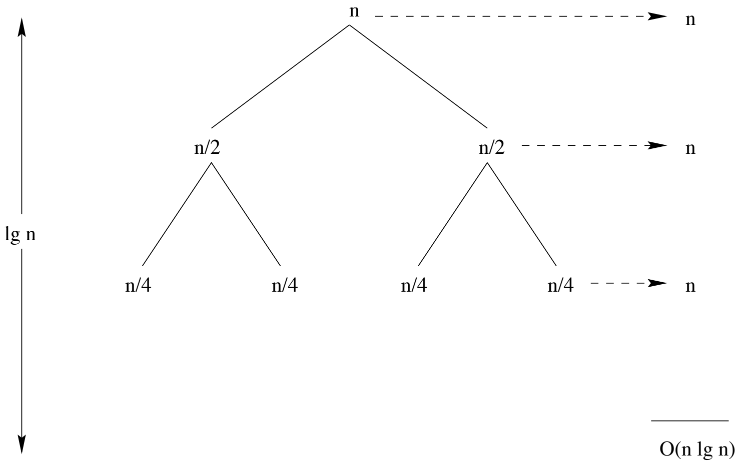
The closed form is $T_{\mathtt{msort}}(n) = \Theta(n \cdot \lg n)$, in all cases.

Merge sort is better than insertion sort in the average and worst cases; insertion sort is better for nearly-sorted data.

Exercise: Redo the analysis for merge sorting an array.

The Recursion-Tree Method

A recursion tree visualises the expansion of a recursion.
(It is used for guessing a closed form, not for proving it.)
The recursion tree for the merge sort recurrence is:



Closing Recurrences

We have already observed that a recurrence of the form

- $T(n) = T(n-1) + \Theta(1)$ gives $\Theta(n)$ (see `sumList`, `ins`).
- $T(n) = T(n-1) + \Theta(n)$ gives $\Theta(n^2)$ (see `isort`).

Divide-and-conquer algorithms give recurrences of the form

$$T(n) = a \cdot T(n/b) + f(n)$$

where 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 theorem.

Proof of the Merge-Sort Recurrence

This proof gives the general flavour of solving divide-and-conquer recurrences.
The formal proof is complicated by technical details, such as when n is not an integer power of b.
We 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 recurrence (4) three pages ago)

$$T(n) = \begin{cases} 2 & \text{if } n = 2 = 2^k \text{ for } k = 1 \\ 2 \cdot T(n/2) + \Theta(n) & \text{if } n = 2^k \text{ for } k > 1 \end{cases} \tag{5}$$

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

UPPSALA UNIVERSITET

Proof by Induction

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

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

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

$$
\begin{aligned}
T(2^{k+1}) &= 2 \cdot T(2^{k+1}/2) + 2^{k+1}, \text{ by the recurrence (5)} \\
&= 2 \cdot T(2^k) + 2^{k+1}, \text{ by arithmetic laws} \\
&= 2 \cdot (2^k \cdot \lg 2^k) + 2^{k+1}, \text{ by the assumption} \\
&= 2^{k+1} \cdot (\lg 2^k + 1), \text{ by arithmetic laws} \\
&= 2^{k+1} \cdot (\lg 2^k + \lg 2^1), \text{ by arithmetic laws} \\
&= 2^{k+1} \cdot \lg 2^{k+1}, \text{ by arithmetic laws} \quad \square
\end{aligned}
$$

Outline

The Master Method and Master Theorem

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

The closed form for a recurrence $T(n) = a \cdot T(n/b) + f(n)$ 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 (known as the 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 (if $f(n) = \Theta(n^{log_b a})$), then $T(n) = \Theta(n^{log_b a} \cdot \lg n)$.
3. If $f(n)$ dominates $n^{\log_b a}$ and if the regularity condition (see the next page) holds, then $T(n) = \Theta(f(n))$.

UPPSALA UNIVERSITET

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 these 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: $n \cdot \lg n$ is not polynomially larger than n.

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

UPPSALA UNIVERSITET

Gaps in the Master Theorem

The Master Theorem does not cover all possible recurrences of the form $T(n) = a \cdot T(n/b) + f(n)$:

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

UPPSALA UNIVERSITET

Common 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	$\Theta(?)$
			$\Theta(n \cdot \lg n)$	3	$\Theta(n \cdot \lg n)$
			$\Theta(n^k)$, 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	$\Theta(n \cdot \lg n)$
			$\Theta(n \cdot \lg n)$	none	$\Theta(?)$
			$\Theta(n^k)$, with $k > 1$	3	$\Theta(n^k)$

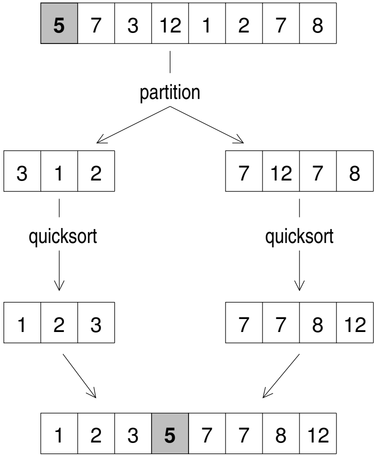
(This table can only be used for looking up a closed form, but it cannot be referred to in the homeworks or exams.)

Outline

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

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

Application of the divide & conquer (& combine) principle:



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
```

Exercise: `partition` (p, L) always takes $\Theta(|L|)$ time.

Quicksort Program


```
sort   L           (* same specification as for merge sort! *)
TYPE: int list             -> int list
PRE:   (none)
POST:  a non-decreasingly sorted permutation of L
EXAMPLE: sort   [5,7,3,12]             = [3,5,7,12        ]

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, but no tail-recursion.

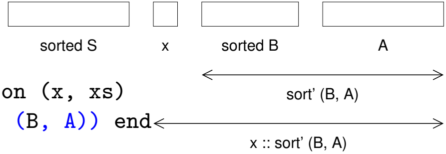
Program uses $X @ Y$, which takes $\Theta(|X|)$ time.

Complexity: `sort` L takes $\Theta(|L| \cdot \lg |L|)$ time on average.

Generalisation by Accumulator Introduction

```
sort' (L, A)
TYPE: int list * int list -> int list
PRE:  (none)
POST: (a non-decreasingly sorted permutation of L) @ A
EXAMPLE: sort' ([5,7,3,12], [1,4,2,3]) = [3,5,7,12,1,4,2,3]

VARIANT: |L|
fun sort' ([], A) = A
  | sort' (x::xs, A) =
    let val (S, B) = partition (x, xs)
    in  sort' (S, x :: sort' (B, A)) end
fun sort L = sort' (L, [])
```



Double recursion, but one tail-recursion.

Program uses no $X@Y$, but the specification of `sort'` does.

Complexity: `sort` L still takes $\Theta(|L| \cdot \lg |L|)$ time on average, but less space; same for `sort'`(L, A), independently of $|A|$.

Worst-Case Analysis

Let $T_{\texttt{qsort}}(n)$ be the time of `sort L` p. 51 on $|L| = n$ elem.s. This naïve version of quicksort always chooses the leftmost element as the pivot. If the list is reverse-sorted, then `partition` always produces lists of $n - 1$ and 0 elements.

Base case ($n = 0$):

Returning `[]` takes $\Theta(1)$ time.

Recursive case ($n > 0$):

- **Divide:** `partition(`x`,`xs`)`, where L `=` x`::`xs, takes $\Theta(|xs|) = \Theta(n - 1) = \Theta(n)$ time, by page 50.
- **Conquer:** The recursive calls take $T_{\texttt{qsort}}(n - 1)$ and $T_{\texttt{qsort}}(0)$ time, as $|S| = n - 1$ and $|B| = 0$ in this case.
- **Combine:** x`::...` takes $\Theta(1)$ time and `(sort` S`)@(..)` takes $\Theta(|S|) = \Theta(n)$ time, by page 16 and $|\texttt{sort } S| = |S|$.

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

Worst-Case Analysis (continued)

Alternatively, let $T_{\texttt{qsort}}(n)$ now be the runtime of `sort L`, computed via `sort'(L,[])` page 52, on $|L| = n$ elements:

Base case ($n = 0$):

Returning the accumulator `A` takes $\Theta(1)$ time.

Recursive case ($n > 0$):

- **Divide:** `partition(x,xs)`, where `L = x::xs`, takes $\Theta(|xs|) = \Theta(n-1) = \Theta(n)$ time, by page 50.
- **Conquer:** The recursive calls take $T_{\texttt{qsort}}(n-1)$ and $T_{\texttt{qsort}}(0)$ time, as $|S| = n-1$ and $|B| = 0$ in this case.
- **Combine:** `x::...` takes $\Theta(1)$ time, by page 16.

The worst-case runtime recurrence for n elements is then:

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

Worst-Case Analysis (end)

Either way, the worst-case runtime recurrence simplifies into:

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

like (3) for insertion sort in the average and worst cases, and hence quicksort takes $\Theta(n^2)$ time in the worst case. (Such reasoning by analogy cannot be done in homeworks).

Exercise: Show, for both quicksort programs, that the case where the list is already sorted (rather than reverse-sorted) also leads to $\Theta(n^2)$ time!

UPPSALA UNIVERSITET

Best-Case Analysis

Let $T_{\mathtt{qsort}}(n)$ be the time of `sort` L page 51 on $|L| = n$ elem.s.
Assume partitioning always produces two equal-length lists, up to one element, their total length being $n - 1$.

Base case ($n = 0$):

Returning `[]` takes $\Theta(1)$ time.

Recursive case ($n > 0$):

- **Divide:** `partition(`x`,`xs`)`, where L `=` x`::`xs, takes $\Theta(|xs|) = \Theta(n-1) = \Theta(n)$ time, by page 50.
- **Conquer:** Each recursive call takes $T_{\mathtt{qsort}}\left(\frac{n-1}{2}\right)$ time, that is $T_{\mathtt{qsort}}\left(\frac{n}{2}\right)$ time, in this case.
- **Combine:** x`::...` takes $\Theta(1)$ time and `(sort` S`)@(..)` takes $\Theta(|S|) = \Theta\left(\frac{n}{2}\right) = \Theta(n)$ time, by page 16 and $|$`sort` $S| = |S|$.

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

Best-Case Analysis (continued)

Alternatively, let $T_{\texttt{qsort}}(n)$ now be the runtime of `sort L`, computed via `sort'(L,[])` page 52, on $|L| = n$ elements:

Base case ($n = 0$):

Returning the accumulator `A` takes $\Theta(1)$ time.

Recursive case ($n > 0$):

- **Divide:** `partition(x,xs)`, where `L = x::xs`, takes $\Theta(|xs|) = \Theta(n-1) = \Theta(n)$ time, by page 50.
- **Conquer:** Each recursive call takes $T_{\texttt{qsort}}\left(\frac{n-1}{2}\right)$ time, that is $T_{\texttt{qsort}}\left(\frac{n}{2}\right)$ time, in this case.
- **Combine:** `x::...` takes $\Theta(1)$ time, by page 16.

The best-case runtime recurrence for n elements is then:

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

Best-Case Analysis (end)

Either way, the best-case runtime recurrence simplifies into:

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

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

UPPSALA UNIVERSITET

Average-Case Analysis

Example: If partitioning always produces 9-to-1 splits (which is a rather poor average behaviour), then the runtime recurrence becomes (for both programs):

$$T_{\text{qsort}}(n) = \begin{cases} \Theta(1) & \text{if } n = 0 \\ T_{\text{qsort}}(9 \cdot n/10) + T_{\text{qsort}}(n/10) + \Theta(n) & \text{if } n > 0 \end{cases}$$

This gives a recursion tree of depth $\log_{10/9} n = \Theta(\lg n)$.

Quicksort takes $\Theta(n \cdot \lg n)$ time in the average case.

Outline

Generalisation by Accumulator Introduction

If for a specification S you have a recursive program P that has no tail-recursion (which is a symptom for unnecessarily high time or space complexity), then design a recursive program P' for the generalised specification S':

S : `f X`
`TYPE:` α `->` β
`PRE:` γ
`POST:` δ

S': `f' (X, A)`
`TYPE:` α `*` β `->` β
`PRE:` γ
`POST:` $\delta \circ$ `A`

where `A` is called an accumulator and it is the function $\circ$ (here in infix notation) used in the combine step that prevents a tail recursion in P.
We get the following non-recursive new program for S:

```
fun f X = f' (X, ε)
```

The old specif. is a specialisation (for $\mathtt{A} = \epsilon$) of the new one.

Requirements on the $\circ$ Function: What Is ϵ?

Program P' exists if $\circ$ is associative ($x \circ (y \circ z) = (x \circ y) \circ z$, for all x, y, z) and has ϵ as right identity element ($x \circ \epsilon = x$, for all x). For instance, $\circ$ can be:

- `@` The left and right identity element is [ ].
  Example: `rev`, quicksort (page 51).
- $+$ The left and right identity element is 0.
  Examples: `length`, `sumUpTo`, and `sumList` (page 7).
- $*$ The left and right identity element is 1.
  Example: `fact`.
- `max` The left and right identity element is $-\infty$ (0, if over $\mathbb{N}$).
  Example: `largest`.
- `min` The left and right identity element is $+\infty$.

but $\circ$ cannot be $-$, `/`, or `div`, which are not associative (and have 0, 1, 1 as right, but not left, identity elements).

Methodology

Methodological Backtracking: If P' is unsatisfactory, but $\circ$ is not commutative ($x \circ y = y \circ x$, for all x, y) and $\circ$ has ϵ as left identity element ($\epsilon \circ x = x$, for all x), then try instead "`POST: A` $\circ\, \delta$" in the specification S'.

Exercise: Redo all accumulator introductions for non commutative $\circ$ in all lecture notes. (Among the associative $\circ$ on the previous page, only `@` is not commutative.)

It is also possible to generalise the program mechanically, but we do not discuss that in this course.

Generalisation is not guaranteed to result in a program with better time or space complexity, hence it is not automatically performed by the SML interpreter or compiler.

Methodology: When & How To Generalise?

There is no need to generalise a specification when the specialised program has optimal time & space complexity!

There exist several specification generalisation techniques:

- Computational generalisation:
  - Descending generalisation: accumulator introduction
  - Ascending generalisation (not covered in this course)
- Structural generalisation (see topic “Binary Trees”)

UPPSALA UNIVERSITET

Outline

Stable Sorting

Two different elements may have equal keys.
Example: When sorting a list of email messages by the day sent, all messages sent on a given day compare “equal”.

Definition

A stable sort never exchanges elements of equal keys.

Example

+ Insertion sort (as above), merge sort (as with the `merge` and naïve `split` above), and quicksort (as with the `partition` and naïve pivot choice above) are stable.
– 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 (page 35).

Outline

Road Map Revisited

Consider various data structures containing n elements:

	insert	delete	search	minimum	...
list	$\Theta(1)$	$O(n)$	$O(n)$	$\Theta(n)$	...
sorted list	$O(n)$	$O(n)$	$O(n)$	$\Theta(1)$	...
sorted array[2]	$O(n)$	$O(n)$	$O(\lg n)$	$\Theta(1)$	...
?	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	...

Table of Contents

- Tree: $O(\lg n)$ time insert, delete, search, minimum, ...
- Stack: $\Theta(1)$ time push (insert) and pop (extractLast).
- First-in first-out queue: $\Theta(1)$ time enqueue, dequeue.
- Heap (priority queue): $O(\lg n)$ time insert, extractMax.
- Hash table: $\Theta(1)$ average time insert, delete, search.

[2] Assuming the indices are not the keys.