

Faculty of Information Technology, Monash University

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act). The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act. Do not remove this notice

FIT2004: Algorithms and Data Structures

Week 1: Introduction

FIT2004 Staff

- **Chief examiner:** Rafael Dowsley
- **Clayton lecturers:** Rafael Dowsley, Yasmeeen George
- **Malaysia lecturer/coordinator:** Lim Wern Han
- **Admin Tutor:** Benjamin Di Stefano
- **Teaching Team:**
 - Alireza Moayedikia
 - Ethan Hunt
 - Harrison Sloan
 - Jackson Goerner
 - Liangrong Zhao
 - Madison Geeson
 - Mubasshir Murshed
 - Nisal Sankalpa De Silva
 - Saman Ahmadi
 - Thomas Hendrey
 - Chit (Roger) Lim
 - Ethan Wills
 - Isobel Nixon
 - Kaleb Barone
 - Luhan Cheng
 - Md Sazzad Hossain
 - Nathan Companez
 - Sachinthana Pathiranage
 - Satya Jhaveri

Contact details can be found on Moodle

What's this unit about

- Solving problems with computers – *efficiently*.
- Developing your algorithm toolbox.
- Training your problem solving skills.
- Deepening your understanding of how the workings of a computer relate to algorithm speed/space usage.

What's this unit about

- The subject *is not* really about *programming*.
- The subject just happens to use Python as the programming language in which assignments are done. This subject is really language agnostic.
不可知的
- Algorithms in this courseware will be presented/described in English, pseudocode, procedural set of instructions or Python (as convenient).

Expectations

- The subject is very important for computer and technology related careers:
 - Companies actively hunt for people good at algorithms and data structures.
 - Many job interview questions are based on algorithms from this unit.
 - Many tutorial questions are very similar to questions you could be asked in job interviews.
 - The things you learn will help you throughout your career.

Expectations

- This unit is CHALLENGING
 - *If you don't have a good understanding of the contents of prerequisite units (e.g., FIT1008), you need to catch up on them urgently, otherwise you will have serious trouble in this unit.*
 - Assignments will test your understanding of the topics as you go.
 - You have to be on top of it from Week 1 – you cannot pass if you think “I can brush up on the material close to the assessment deadlines”.
 - Missing lectures or tutorials will require double the efforts to recover.
- Good news: The unit wants you to succeed and understand. Lot of resources available.

Overview of the content

- **Explore some of the most important algorithm design paradigms:**
 - Divide and Conquer
 - Greedy algorithms
 - Dynamic Programming
 - Network Flow
- **Analysis of algorithms**
- **Learn important data structures for implementing algorithms efficiently**
- **A selection of very important computational problems, such as:**
 - Sorting
 - Retrieval/Lookup
 - Shortest path in graphs
 - Minimum spanning trees

Unit Notes

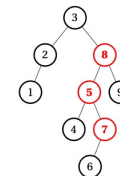
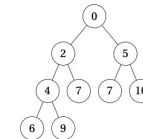
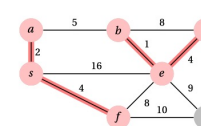
- Course notes are written based on the material covered in the unit.
- Notes for all 12 weeks are available in a single PDF file on Moodle:
 - Click on “Unit Information”
 - Scroll down to “Unit Resources”
 - Click on [“FIT2004 Course Notes”](#)

 MONASH University

FIT2004

Algorithms and Data Structures

The original course notes were developed by Daniel Anderson. Over the course of the years additions, removals, modifications and updates were done by members of FIT2004's teaching teams including: Nathan Companez, Rafael Dowsley.



43242	31311	31311	43122
43122	22241	23411	42143
34344	23411	23312	23143
31311	43242	43122	22541
41423	43122	41423	43242
33444	23332	23332	31311
23332	23312	22241	23312
42143	41423	43242	23332
22241	42143	42143	34344
23143	23143	23143	23411
23411	34344	34344	41423
23312	33444	33444	33444

Last updated February 11, 2022

Additional References

- This subject has immense practical value for your professional development into careers in computer science and information technology. Some books you might want to refer to (from time-to-time, even beyond this unit) include:
 - CLRS: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*.
 - Rou: Tim Roughgarden. *Algorithms Illuminated*.
 - KT: Jon Kleinberg, Éva Tardos. *Algorithm Design*.
 - Knu: Donald Knuth. *The Art of Computer Programming*. (More advanced, pretty expensive; but this is an iconic CS book and very useful for serious programmers; library has copies!)
- **There are no required textbooks. Beware that all of those textbooks contain both less and (far) more than is required for the unit. That is why it is important you do not rely solely on these resources (i.e., you should watch the lectures, attend tutorials and read the course notes).**

Course Material

- Your main portal will be, as you already know, the unit's Moodle page
- Material available on Moodle will include:
 - Lecture slides
 - Lecture recordings
 - Assignments
 - Course notes
 - Tutorial sheets and solutions
- Ed Discussion Platform (Not Moodle forums!)
 - General Q & A
 - Assignment amendments
 - Lecture error correction
 - Changes to consultation time
 - etc

Give us feedback!

I imagine that as the unit runs, most of you will have things you dislike/think should be done differently.

Tell us about it!

- Send us an email.
- Post (privately) on Ed.

A few examples of changes which were motivated by student feedback:

- Reduction in exam percentage to 50%.
- Reduction of the number of assignments from 4 to 2.
- Forum section specifically for sharing test cases for assignments.
- Releasing tutorial solutions one week earlier.

Assessment Summary

- Quiz 10%
- Assignments 1-2 40%
 - Two practical assignments (each worth 20% each) focusing on implementing algorithms *satisfying certain complexity requirements*.
 - **Must** be implemented in Python. Start early!!!
- Final Exam 50%

Hurdles

- To pass this unit a student must obtain:
 - **45% or more in the unit's final exam (i.e., at least 22.5 out of 50 marks for final exam), and**
 - **45% or more in the unit's total non-examination assessment (i.e., at least 22.5 out of total 50 marks for quiz and assignments), and**
 - **an overall unit mark of 50% or more.**

How to approach assignments

- Historically, many students have good in semester scores but fail the exam.
- Many students focus all their effort in this unit on completing assignments, but the assignments are designed to be done by people with a strong grasp of the content and are heavily based on the content taught in the unit.
- If you spend time understanding the content, the assignments will be far easier.
- So will the exam!

Cheating, Collusion, Plagiarism

- **Cheating:** Seeking to obtain an unfair advantage in an examination or in other written or practical work required to be submitted or completed for assessment.
- **Collusion:** Unauthorised collaboration on assessable work with another person or persons.
- **Plagiarism:** To take and use another person's ideas and or manner of expressing them and to pass them off as one's own by failing to give appropriate acknowledgement. This includes material from any source, staff, students or the Internet – published and un-published works.
- **The use of generative AI and similar tools is not allowed in this unit!**

<http://infotech.monash.edu.au/resources/student/assignments/policies.html>

How to avoid collusion for FIT2004

- Do not discuss the assignment with other students.
- ***High-risk, low-gain game: last semester academic integrity cases were open to investigate 9% of the cohort. After SCC investigations and decisions, almost all those students got either a “zero marks for assignment” penalty (and consequently very few were able to pass the unit hurdle) or a straight “zero marks for unit” penalty.***
- Most academic integrity cases in the past turned out to be:
 - I was trying to help my friend
 - I gave them my code so they would have the right idea, I didn't think they would use it
 - I let them look at my code, but they took a photo of it and copied it
 - etc...
- What can you do? Share test cases! Feel free to post your test cases on the Ed post that will be created for that, and to use other people's.

1st Algorithm

- What was the first algorithm you learned?
 - As an IT student you already learned some algorithms in the university: binary search, sorting algorithms, etc.
 - But algorithms predate computers by millennia (even the word ‘algorithm’ is derived from the name of a 9th century Persian mathematician).
 - In fact, you learned in school an algorithm that is more than 2000 years old: Euclid’s algorithm for computing the greatest common divisor.
 - Even in your first school years you already learned the “grade school” multiplication algorithm.

Multiplication Algorithm

$$\begin{array}{r} \times \quad 123 \\ 345 \\ \hline 615 \\ 492 \\ 369 \\ \hline 42435 \end{array}$$

- Grade school multiplication algorithm using partial products.
- Did your teacher talk about the efficiency of this algorithm? Showed the correctness proof?
- Well, back then you were only a user of the algorithm. **In the future, understanding the efficiency and correctness of algorithms will be a central skill in your career.**

Multiplication Algorithm

$$\begin{array}{r} 123 \\ \times 345 \\ \hline 615 \\ 492 \\ 369 \\ \hline 42435 \end{array}$$

- If we consider addition and multiplication of single digit numbers as the basic operations, for n -digit numbers, this algorithm clearly has complexity $O(n^2)$.
- Fundamental question in algorithm design: Can we do it more efficiently?

Outline

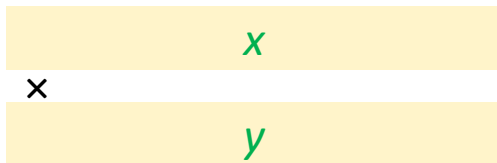
- Divide-and-Conquer
- Complexity Analysis
- Solving Recurrence Relations

Divide-and-Conquer Paradigm

- The divide-and-conquer algorithm design paradigm works in 3 steps:
 1. *Divide* the problem into smaller subproblems.
 2. *Conquer* (i.e., solve) the smaller subproblems.
 3. *Combine* the solutions of the smaller subproblems to obtain the solution of the bigger problem.
- Analysing the time complexity of a divide-and-conquer algorithm normally involves solving a recurrence relation (studied later in this lecture).
- Normally a polynomial time solution to the problem is already known, and the divide-and-conquer strategy is used to reduce the time complexity to a lower polynomial.

First Improvement Idea

- Problem: multiply two n -digits numbers x and y in sub-quadratic time given addition and multiplication of single digit numbers as the basic operations.



A diagram illustrating a multiplication problem. It shows two horizontal yellow bars. The top bar contains the variable x in green. The bottom bar contains the variable y in green. To the left of the bottom bar, there is a small black 'x' symbol, indicating multiplication.

- Improvement idea: Split the numbers between the $n/2$ most significant digits and the $n/2$ least significant digits; and do recursive calls with them.

First Improvement Idea

- Problem: multiply two n -digits numbers x and y in sub-quadratic time given addition and multiplication of single digit numbers as the basic operations.

	x_M	x_L
\times	y_M	y_L

- Improvement idea: Split the numbers between the $n/2$ most significant digits and the $n/2$ least significant digits; and do recursive calls with them.
- From math we know that:

$$\begin{aligned}x \cdot y &= (x_M \cdot 10^{n/2} + x_L) (y_M \cdot 10^{n/2} + y_L) \\&= x_M \cdot y_M \cdot 10^n + x_M \cdot y_L \cdot 10^{n/2} + x_L \cdot y_M \cdot 10^{n/2} + x_L \cdot y_L\end{aligned}$$

First Improvement Idea

- We have that $x \cdot y = x_M \cdot y_M \cdot 10^n + x_M \cdot y_L \cdot 10^{n/2} + x_L \cdot y_M \cdot 10^{n/2} + x_L \cdot y_L$
- Reduce 1 instance of the problem of size n to 4 instances of size $n/2$:

x_M	x_M	x_L	x_L
×	×	×	×
y_M	y_L	y_M	y_L

- Are we making progress?
- Not really! Intuition: 4 instances that will each take about $\frac{1}{4}$ of time that was necessary to solve the original problem, so the overall time stays in the same order. You can later check that by solving the recurrence $T(n) \leq 4T(n/2) + c \cdot n$.
- If we want to follow this approach to improve the efficiency, we should use at most 3 recursive calls.

Are Improvements possible?

Conjecture: Sub-quadratic multiplication is impossible!



Andrey Kolmogorov, one of the greatest mathematicians of the 20th century.

That is wrong!



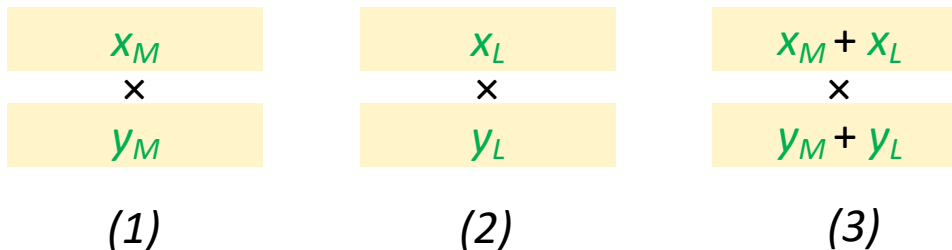
Anatoly Karatsuba, then a 23 y/o student, within one week from hearing Kolmogorov's conjecture in his seminar.

Karatsuba's Algorithm

- We have that:

$$\begin{aligned}x \cdot y &= x_M \cdot y_M \cdot 10^n + x_M \cdot y_L \cdot 10^{n/2} + x_L \cdot y_M \cdot 10^{n/2} + x_L \cdot y_L \\&= x_M \cdot y_M \cdot 10^n + (x_M \cdot y_L + x_L \cdot y_M) \cdot 10^{n/2} + x_L \cdot y_L\end{aligned}$$

- Do only 3 recursive calls to compute:



- Given the results of (1), (2) and (3), if we can trivially obtain $(x_M \cdot y_L + x_L \cdot y_M)$ then we are done computing $x \cdot y$ with only 3 recursive calls.

Karatsuba's Algorithm

- **Problem:** Obtain $z = (x_M \cdot y_L + x_L \cdot y_M)$ without further recursive calls when given:

x_M	x_L	$x_M + x_L$
\times	\times	\times
y_M	y_L	$y_M + y_L$
(1)	(2)	(3)

- Note that

$$\begin{aligned}(x_M + x_L)(y_M + y_L) &= x_M \cdot y_M + x_M \cdot y_L + x_L \cdot y_M + x_L \cdot y_L \\ &= z + x_M \cdot y_M + x_L \cdot y_L\end{aligned}$$

- **Solution:** To obtain z we can simply subtract the result of the 1st and 2nd recursive calls from the result of the 3rd recursive call.
- Where does this trick come from?

Karatsuba's Algorithm

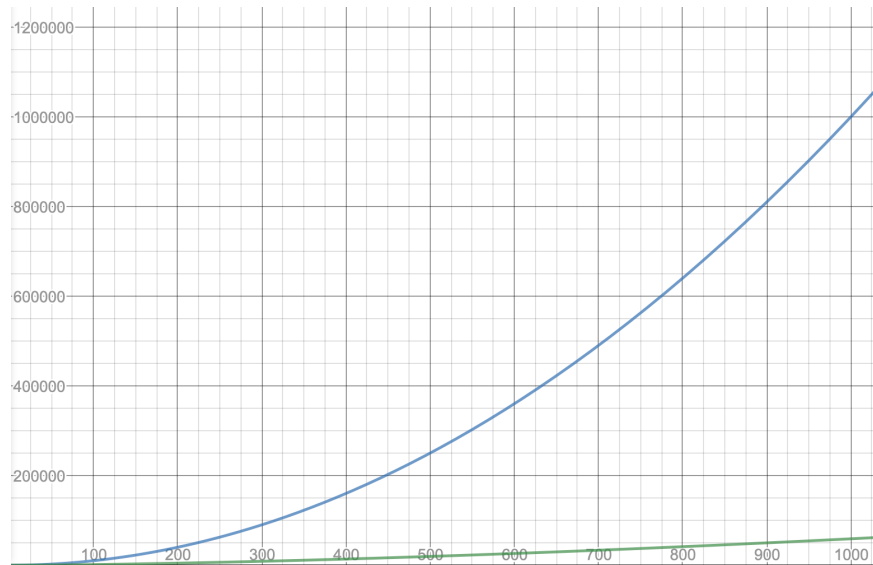
- This trick traces back to the method developed by Gauss to multiply complex numbers using 3 multiplications of real numbers instead of 4.
- **Adapting previous ideas to solve your new problem can be very useful!**



Johann Carl Friedrich Gauss

Karatsuba's Algorithm

- This is the algorithm that Python uses for multiplying large numbers.
- The time complexity of Karatsuba's algorithm is $O(n^{1.59})$. To verify that, just solve the recurrence $T(n) \leq 3T(n/2) + c \cdot n$ and use the fact that $\log_2 3 < 1.59$.
- Don't underestimate the difference between n^2 (blue line) and $n^{1.59}$ (green)!



Mergesort (Revision)

- One of the first explicit uses of the divide-and-conquer paradigm.
- $O(n \log n)$ sorting algorithm presented by John von Neumann.
- Python uses Timsort (a hybrid stable sorting algorithm derived from Mergesort and insertion sort) as standard.

Algorithm 5 Merge sort

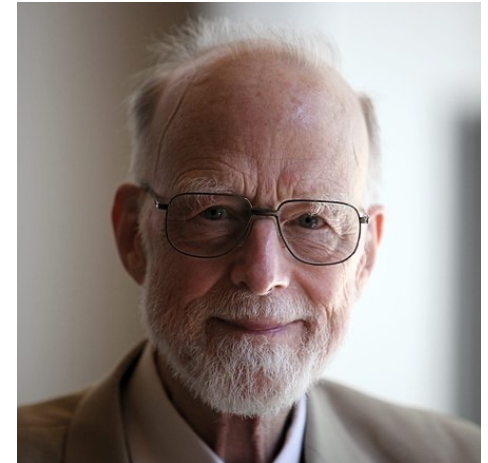
```
1: function MERGE_SORT(array[lo..hi])
2:   if hi > lo then
3:     mid =  $\lfloor (lo + hi) / 2 \rfloor$ 
4:     MERGE_SORT(array[lo..mid])
5:     MERGE_SORT(array[mid + 1..hi])
6:     array[lo..hi] = MERGE(array[lo..mid], array[mid + 1..hi])
```



John von Neumann, one of the greatest mathematicians of the 20th century, his Wiki page “known for” has 100 items!

Quicksort

- Another sorting algorithm that uses the divide-and-conquer paradigm, but the subproblems are not necessarily of the same size. Created by Tony Hoare.
- Lecture 3 will cover it in detail.



Tony Hoare
(Turing Award 1980)

Other Divide-and-Conquer Examples

The divide-and-conquer paradigm can be applied to get efficient algorithms for a wide range of problems, such as:

- Finding closest pair of points in a plane in $O(n \log n)$.
- Counting inversions in $O(n \log n)$, see Tutorial 2.
- Improving matrix multiplication (Strassen's algorithm).
- Fast Fourier Transform: this algorithm published by James Cooley and John Tukey in 1965 is one of the most influential algorithms, with a wide range of applications in engineering, music, science, mathematics, etc.
 - In fact, it can be traced back to unpublished work by Gauss.

Outline

- Divide-and-Conquer
- Complexity Analysis $O(m/k) = \log_k(m)$ $m = 4, k = 2, \log_2(4) = 2$
 $O(m/2)$ half input size everytime
- Solving Recurrence Relations

Complexity Analysis

- **Time complexity** is the amount of time taken by an algorithm to run as a function of the input size.
- Worst-case complexity (our main focus)
- Best-case complexity
- Average-case complexity

Complexity Analysis

- **Big-O notation:** $f(n) = O(g(n))$ if there are constants c and n_0 such that $f(n) \leq c \cdot g(n)$, for all $n \geq n_0$.
- **Big-Ω notation:** $f(n) = \Omega(g(n))$ if there are constants c and n_0 such that $f(n) \geq c \cdot g(n)$, for all $n \geq n_0$.
- **Big-Θ notation:** $f(n) = \Theta(g(n))$ if, and only if, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Complexity Analysis

- **Space complexity** is the total amount of space taken by an algorithm as a function of input size.
- **Auxiliary space complexity** is the amount of space taken by an algorithm **excluding** the space taken by the input.
- Many textbooks and online resources do not distinguish between the above two terms and use the term “space complexity” when they are in fact referring to auxiliary space complexity. In this unit, we use these two terms to differentiate between them.

In-place Algorithm

- **In-place algorithm:** An algorithm that has $O(1)$ auxiliary space complexity.
 - i.e., it only requires constant space in addition to the space taken by input
 - Merging is not an in-place algorithm as it needs to create the output list which is size N .
 - Be mindful that some books use a different definition (e.g., space taken by recursion may be ignored). For the sake of this unit, we will use the above definition.

Space Complexity: Finding minimum

//Find minimum value in an unsorted array of $N > 0$ elements

```
min = array[1]
```

```
index = 2
```

```
while index <= N
```

```
    if array[index] < min
```

```
        min = array[index]
```

```
    index = index + 1
```

```
return min
```

- Space complexity?
 - $O(N)$
- Auxiliary space complexity?
 - $O(1)$
- This is an in-place algorithm.

Time/Space Complexity: Binary Search

```
lo = 1
hi = N
while ( lo <= hi )
    mid = floor( (lo+hi)/2 )
    if key == array[mid]
        return mid
    if key >= array[mid]
        lo=mid+1
    else
        hi=mid-1
return False
```

Time Complexity?

- Worst-case

- Search space at start: N
- Search space after 1st iteration: $N/2$
- Search space after 2nd iteration: $N/4$
- ...
- Search space after x-th iteration: 1

What is x? i.e., how many iterations in total?

$O(\log N)$

- Best-case

- $O(1)$, returning key when $\text{key} == \text{array}[\text{mid}]$

Space Complexity?

- $O(N)$

Auxiliary Space Complexity?

- $O(1)$

Binary search is an in-place algorithm!

Output-Sensitive Time Complexity

Problem: Given a sorted array of unique numbers and two values x and y , find all numbers greater than x and smaller than y .

Algorithm 1:

For each number n in array:

 if $n > x$ and $n < y$:

 print(n)

Time complexity:

$O(N)$

Output-Sensitive Time Complexity

Problem: Given a sorted array of unique numbers and two values x and y , find all numbers greater than x and smaller than y .

Algorithm 2:

- Binary search to find the smallest number greater than x .
- Continue linear search from x until next number is $\geq y$.

Time complexity?

$O(N)$ in the worst-case because in the worst-case all numbers may be within the range x to y .

flux.qa/F7CWW9



Output-Sensitive Time Complexity

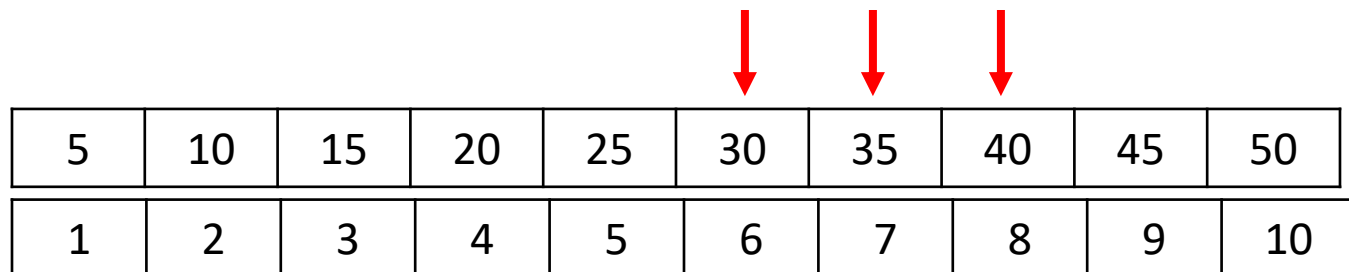
Output-sensitive complexity is the time-complexity that also depends on the size of output. Let W be the number of values in the range (i.e., in output).

Output-sensitive complexity of Algorithm 2? $O(W + \log N)$ – note W may be N in the worst-case.

Output-sensitive complexity of Algorithm 1? $O(N)$

Output-sensitive complexity is only relevant when output-size may vary, e.g., it is not relevant for sorting, finding minimum value etc.

Assume $x=28$ and $y=38$



5	10	15	20	25	30	35	40	45	50
1	2	3	4	5	6	7	8	9	10

Outline

- Divide-and-Conquer
- Complexity Analysis
- Solving Recurrence Relations

Recurrence Relations

A recurrence relation is an equation that recursively defines a sequence of values, and one or more initial terms are given, e.g.,

$$T(1) = b$$

$$T(N) = T(N-1) + c$$

- Complexity of recursive algorithms can be analysed by writing its recurrence relation and then solving it.

Solving Recurrence Relations

// Compute Nth power of x

power(x,N)

{

if (N==0)

return 1

if (N==1)

return x

else

return x * power(x, N-1)

}

Time Complexity

Cost when $N = 1$: $T(1) = b$ (b&c are constant)

Cost for general case: $T(N) = T(N-1) + c$ (A)

Cost for N-1: $T(N-1) = T(N-2) + c$

Replacing $T(N-1)$ in (A)

$T(N) = (T(N-2) + c) + c = T(N-2) + 2*c$ (B)

Cost for N-2: $T(N-2) = T(N-3) + c$

Replacing $T(N-2)$ in (B)

$T(N) = T(N-3) + c+c+c = T(N-3) + 3*c$

Our goal is to reduce this term to $T(1)$

Do you see the pattern?

$T(N) = T(N-k) + k*c$

Solving Recurrence Relations

// Compute Nth power of x

power(x,N)

{

if (N==0)

return 1

if (N==1)

return x

else

return x * power(x, N-1)

}

Time Complexity $T(N=1) = b$

Cost when $N = 1$: $T(1) = b$ (b&c are constant)

Cost for general case: $T(N) = T(N-1) + c$ (A)

$1 = 0 + c, c = 1$ reduce it to $T(1)$

TC of X, just multiplication, so complexity add up

$T(N) = T(N-k) + k*c$

therefore

Find the value of k such that $N-k = 1 \rightarrow k = N-1$

so $T(N-k) = T(1)$

$T(N) = T(N-(N-1)) + (N-1)*c = T(1) + (N-1)*c$

$T(N) = b + (N-1)*c = c*N + b - c$

since c is constant, to get $T(N)$ in terms of c

Hence, the complexity is $O(N)$

Solving Recurrence Relations

```
// Compute Nth power of x
power(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    else
        return x * power(x, N-1)
}
```

Time Complexity

Cost when $N = 1$: $T(1) = b$ (b & c are constant)

Cost for general case: $T(N) = T(N-1) + c$ (A)

$$T(N) = c * N + b - c$$

Check by substitution:

$$\begin{aligned} T(N-1) + c &= c * (N-1) + b - c + c \\ &= c * (N-1) + b \\ &= c * N + b - c \\ &= T(N) \end{aligned}$$

As required

$$T(1) = c * 1 + b - c = b$$

As required

Space Complexity of Recursive Algorithms

// Recursive version

```
power(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    else
        return x * power(x, N-1)
}
```

// Iterative version

```
result = 1
for i=1; i<= N; i++){
    result = result * x
}
return result
```

Space Complexity?

Total space usage = local space used by the function * maximum depth of recursion

every stack has own space and version to store variable value

= c * maximum depth of recursion = c*N

= O(N)

return result as parameter,
the stack is gone, so O(1)

Note: We will not discuss tail-recursion in this unit because it is language specific, e.g., Python doesn't utilize tail-recursion

T(n) calling initial function, T(N-1) call another time with a smaller input

Auxiliary Space Complexity?

- Recursive power() is not an in-place algorithm
tail-recursion like it, just return one result

Note that an iterative version of power uses O(1) space and is an in-place algorithm

Recurrence and complexity

Recurrence relation:

$$T(N) = T(N/2) + c$$

$$T(1) = b$$

Example algorithm?

Binary search

Solution:

$$O(\log N)$$

Recurrence and complexity

Recurrence relation:

$$T(N) = T(N-1) + c$$

$$T(1) = b$$

Example algorithm?

Linear search

Solution:

$$O(N)$$

Recurrence and complexity

Recurrence relation:

$$T(N) = 2 * T(N/2) + c * N$$

$$T(1) = b$$

Example algorithm?

Mergesort

Solution:

$$O(N \log N)$$

Recurrence and complexity

Recurrence relation:

$$T(N) = T(N-1) + c * N$$

$$T(1) = b$$

Example algorithm?

Selection sort

Solution:

$$O(N^2)$$

Recurrence and complexity

Recurrence relation:

$$T(N) = 2 * T(N-1) + c$$

$$T(0) = b$$

Example algorithm?

Naïve recursive Fibonacci

Solution:

$$O(2^N)$$

More about recurrences in Tutorial 2 next week!

Reading

- Course Notes: Sections 1.2, 1.3 and 1.4, Chapter 2
- Additional resources (not required, but also not necessary covering all topics): contents related to this lecture and recap of previous units can be found in standard algorithms books, e.g.:
 - CLRS: Chapters 3 and 4
 - KT: Chapters 2 and 5
 - Rou: Chapters 1 to 4

Concluding Remarks

Summary

- This unit demands your efforts from Week 1.
- The divide-and-conquer algorithm design paradigm can be useful for reducing the complexity.

Coming Up Next

- Analysis of algorithms
- Non-comparison based sorting (Counting Sort, Radix Sort)

IMPORTANT: Preparation required before the next week

- Revise computational complexity covered in earlier units (FIT1045, FIT1008).
- **Complete Tutorial 1 (in your own time).**