Overview

I. Introduction and getting started
II. Growth of functions and asymptotic notations
III. Divide-and-conquer recurrences and the master theorem
IV. Divide-and-conquer algorithms
V. Greedy algorithms
VI. Dynamic programming
VII. Graph algorithms
VIII. NP-completeness

Based on Chapters 1-4, 15-16, 22-25 and 34-35 of the textbook.
I. Introduction and Getting Started
Introduction

- Algorithm is a tool for solving a well-specified computational problem
Introduction

- Algorithm is a tool for solving a well-specified computational problem
- An algorithm is a well-defined procedure for transforming some input into a desired output
Introduction

- Algorithm is a tool for solving a well-specified computational problem
- An algorithm is a well-defined procedure for transforming some input into a desired output

- A poem by D. Berlinski in “Advent of the Algorithm”

  *In the logician’s voice:*

  *an algorithm is*
  *a finite procedure,*
  *written in a fixed symbolic vocabulary*
  *governed by precise instructions,*
  *moving in discrete steps, 1, 2, 3, ...*
  *whose execution requires no insight, cleverness,*
  *intuition, intelligence, or perspicuity*
  *and that sooner or later comes to an end.*
Introduction

- Algorithm is a tool for solving a well-specified computational problem
- An algorithm is a well-defined procedure for transforming some input into a desired output
- A poem by D. Berlinski in “Advent of the Algorithm”

In the logician’s voice:

an algorithm is
a finite procedure,
written in a fixed symbolic vocabulary
governed by precise instructions,
moving in discrete steps, 1, 2, 3, ...
whose execution requires no insight, cleverness, intuition, intelligence, or perspicuity
and that sooner or later comes to an end.

- Algorithms as a technology

How Algorithms Shape Our World, a TED talk by Kevin Slavin
Introduction

- Basic questions about an algorithm
  1. Does it halt?
  2. Is it correct?
  3. Is it fast? (Can it be faster?)
  4. How much memory does it use?
  5. How does data communicate?
Getting started: example 1

- Fibonacci numbers:

\[
F_0 = 0, \\
F_1 = 1, \\
F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2
\]
Getting started: example 1

- Fibonacci numbers:
  \[ F_0 = 0, \]
  \[ F_1 = 1, \]
  \[ F_n = F_{n-1} + F_{n-2} \quad \text{for} \quad n \geq 2 \]

- Fibonacci numbers grow *almost* as fast as the power of 2:
  \[ F_n \approx 2^{0.694n} \]
Getting started: example 1

▶ Fibonacci numbers:

\[ F_0 = 0, \]
\[ F_1 = 1, \]
\[ F_n = F_{n-1} + F_{n-2} \quad \text{for} \quad n \geq 2 \]

▶ Fibonacci numbers grow \emph{almost} as fast as the power of 2:

\[ F_n \approx 2^{0.694n} \]

▶ Problem statement:

\emph{computing the} \( n \)-th \emph{Fibonacci number} \( F_n \)
Getting started: example 1

- Fibonacci numbers:
  \[
  F_0 = 0, \\
  F_1 = 1, \\
  F_n = F_{n-1} + F_{n-2} \quad \text{for} \quad n \geq 2
  \]

- Fibonacci numbers grow \textit{almost} as fast as the power of 2:
  \[
  F_n \approx 2^{0.694n}
  \]

- Problem statement:
  \textit{computing the }n\text{-th Fibonacci number }F_n\text{.}

- Algorithms for computing the }n\text{-th Fibonacci number }F_n:\n  1. Recursion ("top-down")
  2. Iteration ("bottom-up", memoization)
  3. Divide-and-conquer
  4. Approximation
Getting started: example 2

▶ Problem statment:

Input: a sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: a permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the $a$-sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

In short, sorting
Getting started: example 2

- Problem statement:
  
  **Input:** a sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  
  **Output:** a permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the $a$-sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

  In short, *sorting*

- Algorithms:
  
  1. Insertion sort
  2. Merge sort
Getting started: example 2

Insert sort algorithm

- Idea: incremental approach
- Pseudocode

```plaintext
InsertionSort(A)
1 n = length(A)
2 for j = 2 to n
3    key = A[j]
4    // insert 'key' into sorted array A[1...j-1]
5    i = j-1
6    while i > 0 and A[i] > key do
7       A[i+1] = A[i]
8       i = i-1
9    end while
10   A[i+1] = key
11 end for
12 return A
```
Getting started: example 2

Remarks:

▶ Correctness: argued by “loop-invariant” (a kind of induction)
▶ Complexity analysis: let $T(n)$ be the number of operations for sorting an array of length $n$, and $t_j$ be the number of while-loop executed for $j$, then

$$T(n) = \sum_{j=2}^{n} (1 + 1 + t_j + 1) = 3(n - 1) + \sum_{j=2}^{n} t_j$$

▶ best-case: $t_j = 1$ and $T(n) = 4(n - 1) = O(n)$
▶ worst-case: $t_j = j$ and $T(n) = 3(n - 1) + \sum_{j=2}^{n} j = O(n^2)$
▶ average-case: $t_j = \frac{j}{2}$ and $T(n) = 3(n - 1) + \sum_{j=2}^{n} \frac{j}{2} = O(n^2)$

▶ Insertion sort is a “sort-in-place”, no extra memory necessary
▶ Importance of writing a good pseudocode = “expressing algorithm to human”
▶ There is a recursive version of insertion sort, see Homework 1.
Getting started: example 2

Merge sort algorithm

- Idea: divide-and-conquer approach
- Pseudocode

```
MergeSort(A,p,r) // Merge-sort of array A[p..r]
1 if p < r then // check for base case
2     q = flooring( (p+r)/2 ) // divide
3     MergeSort(A,p,q) // conquer
4     MergeSort(A,q+1,r) // conquer
5     Merge(A,p,q,r) // combine
6 end if
```
Getting started: example 2

- Pseudocode, cont’d

```
Merge(A,p,q,r)
  n1 = q-p+1;  n2 = r-q
  for i = 1 to n1 // create arrays L[1...n1+1] and R[1...n2+1]
    L[i] = A[p+i-1]
  end for
  for j = 1 to n2
    R[j] = A[q+j]
  end for
  L[n1+1] = infty; R[n2+1] = infty // mark the end of arrays L and R
  i = 1; j = 1
  for k = p to r // Merge arrays L and R to A
    if L[i] <= R[j] then
      A[k] = L[i]
      i = i+1
    else
      A[k] = R[j]
      j = j+1
    end if
  end for
```
Getting started: example 2

- Merge sort is a divide-and-conquer algorithm consisting of three steps: divide, conquer and combine.
- To sort the entire sequence $A[1...n]$, we make the initial call
  $$\text{MergeSort}(A,1,n)$$
  where $n = \text{length}(A)$.
- Complexity analysis:
  $$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n - 1 = O(n \lg(n))$$