Data Structures – LECTURE 1

Introduction

- Motivation: algorithms and abstract data types
- Easy problems, hard problems
- Examples: sorting
- Course contents

Programs and algorithms

- Why do we need algorithms?
  → to solve problems with a computing device
- What is the difference between an algorithm and a program?
  → a program is an *implementation* of an algorithm to be run on a specific computer and operating system.
  → an algorithm is more abstract – it does not deal with machine specific details – think of it as a *method* to solve a problem.
- The course emphasis is on <u>algorithms.</u>

Data structures

- A data structure is a method of storing data for the purpose of efficient computation
  → *variables, arrays, linked lists, binary trees*
- How data is stored is key for how a problem will be solved.
- Assumptions about the data determine what data structure and algorithm will be used
  → sorting *integers* vs. *words*
- Data structures and algorithm development go together! You cannot have one without the other!

Abstract Data Types –ADT

- An abstract data type is a collection of <u>formal specifications</u> of data-storing entities with a well designed set of operations.
- The set of operations defined with the ADT specification are the operations it "supports"
- What is the difference between a data structure (or a class of objects) and an ADT?
  → The data structure or class is an *implementation* of the ADT to be run on a specific computer and operating system. Think of it as an abstract JAVA class. The course emphasis is on <u>ADTs.</u>

Focus of the course

- In this course, we will study algorithms and ADTs for solving the most common computational problems:

  searching, sorting, indexing, ...
- We will learn how to rigorously analyze an algorithms in terms of <u>*space and time complexity*</u>

  → is A_1 always better than A_2?
- We will learn how to adapt know algorithms and develop new ones.

Algorithms and problem solving

Say you have a computational problem to solve

- Is there an algorithm that solves it?
  → not always! Example: the halting problem.
- Is there an <u>efficient</u> algorithm that solves it?
  → not always! Example: packing problem.
- Is my algorithm the best possible algorithm?
  → not necessarily! Example: sorting in $O(n^2)$
- What is the best algorithm we can develop?
  → sorting takes $\Omega(n\log n)$ time and $\Omega(n)$ space.

Easy problems, hard problems

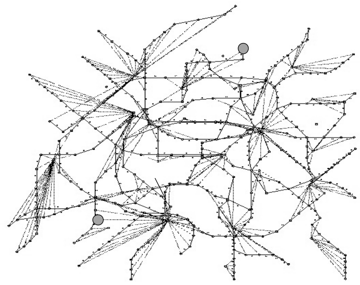
- Over the past 50 years (and especially the last 30 years), many algorithms for a wide variety of computational tasks have been developed
- A classification of hard and easy problems has also been developed, together with formal tools to prove what is their complexity and how they are related to each other.

→ Equivalence classes of *complexity*

- $\Omega(n)$ – linear; $\Omega(n\log n)$;
- $\Omega(n^2)$ – quadratic; $\Omega(n^k)$ – polynomial;
- $\Omega(2^n)$ – exponential; $\Omega(2^{2^n})$ doubly exponential
- unsolvable!

Easy problem: shortest path planning

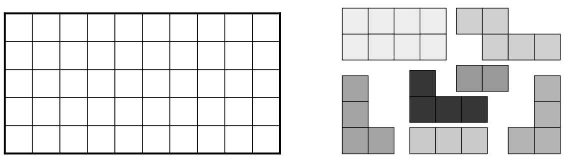
Find the shortest path (minimum number of changes and stops) between two stations in the Paris metro



$O(n)$
in the number
of segments

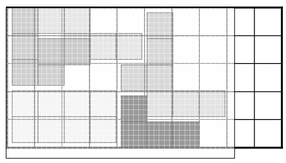
Bin packing: a hard problem!

Given a board and a set of parts, pack them without overlap so that they occupy the smallest rectangle.

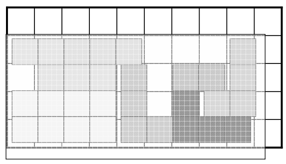


7 parts, 30 squares

Bin packing: possible solutions

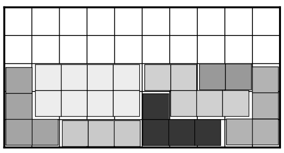


40 squares



36 squares

Bin packing: optimal solution



30 squares

Algorithm

1. generate all legal combinations
2. record area covered
3. keep the best one

Number of legal configurations: combinatorial

There is no better solution in the worst case!!

What kind of efficiency?

Given an algorithm *A*, we can ask the following questions on its time and space complexity.

- Best case: what is the complexity for the ***most*** favorable kind of input?
- Worst case: what is the complexity for the ***least*** favorable kind of input?
- Average case: what is the complexity for the ***average*** kind of input?
- Upper and lower bounds: what is the best we can do for this problem?
- Trade-off between time and space.

Efficiency: sorting

Given an array of n integers A[i], sort them in increasing order.

Two algorithms (among many others) to do this:

- <u>BubbleSort</u>: compare two adjacent numbers, and exchange them if A[i-1] < A[i]. Repeat *n* times.
- <u>MergeSort</u>: recursively split the array in half, sort each part, and then merge them together.

Bubble sort (1)

84	55	61	10	18	35	22	97	47	***start***
55	**84**	61	10	18	35	22	97	47	1st iteration
55	61	**84**	10	18	35	22	97	47	2nd iteration
…									…
55	61	10	18	35	22	97	84	**97**	nth iteration

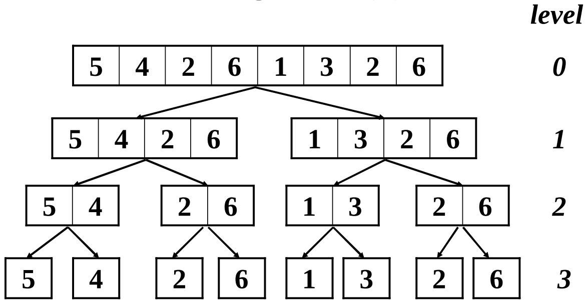
One pass

Bubble sort (2)

55	61	10	18	35	22	97	84	**97**	1st pass
55	10	18	35	22	61	47	**84**	**97**	2nd pass
10	18	35	22	55	61	**22**	**84**	**97**	3rd pass
…									…
10	18	22	35	47	55	61	84	97	nth pass

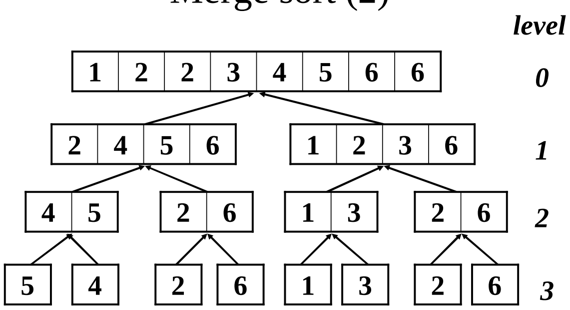
n passes

Merge sort (1)



Split phase

Merge sort (2)



Merge phase

Comparison

	SPACE	T	I M	E
		Best	Worst	Average
Bubble Sort	*n*	*n* *one pass*	n^2 *n passes*	$n^2/2$ *n/2 passes*
Merge Sort	*n* log *n*	*n* log *n*	*n* log *n*	*n* log *n*

MergeSort:
- Number of levels: $2^l = n \rightarrow l = \log_2 n$
- Time for merge: *n*

Other types of algorithms and analyses

Up to now, you have studied exact, deterministic algorithms. There are other types as well:

- Randomized algorithms: makes random choices during execution: pick a random element from an array instead of the first one → minimize the chances of always picking a bad one!
- Probabilistic analysis for randomized algorithms
- Approximation algorithms: instead of finding an optimal solution, find one close to it → bin packing.

Course topics (1)

- Techniques for formal analysis of asymptotic algorithm complexity with recurrence equations
- Techniques for solving recurrence equations: substitution, recursion-tree, master method.
- Proving upper and lower bounds
- Sorting, in-depth: merge sort, quick sort, counting sort, radix sort, bucket sort.

Course topics (2)

- Common ADTs and their algorithms: heaps, priority queues, binary trees, AVL trees, B-trees
- Hash tables and hash functions
- Graph algorithms: Breadth-First Search, Depth-First Search, Shortest path algorithms, Minimum Spanning Trees, Strongly Connected Components.
- Union-Find of sets (time permitting).