**Plan: Amortization and Problem Classification**

**Objective**

## In this lecture, the students should be introduced to amortization – techniques of analysis and amortized bound, P vs NP – heuristic algorithms, general problem solving techniques – problem solving, five steps for "easy" DP, generic graph algorithms and conclusion.

**Motivation**

Amortized analysis is a powerful technique for performance analysis, involving the total runtime of a sequence of operations, which is often what we really care about. The main advantage of adopting a heuristic approach is that it offers a quick solution, which is easy to understand and implement. Heuristic algorithms are practical, serving as fast and feasible short-term solutions to planning and scheduling problems.

**Content**

### Amortization

* Amortized analysis is a powerful technique for performance analysis, involving the total runtime of a sequence of operations, which is often what we really care about.
* The motivation for amortized analysis is that looking at the worst-case run time per operation, rather than per algorithm, can be too pessimistic
* Diﬀerent techniques of amortized analysis
  + Aggregate method
  + Accounting method
  + Charging method
  + Potential method
* The method we used in the previous analysis is the aggregate method – just add up the cost of all the operations and then divide by the number of operations.

amortized cost per operation = total cost of k operations / k

* Aggregate method is the simplest method. Because it’s simple, it may not be able to analyze more complicated algorithms.
* This method allows an operation to store credit into a bank for future use, if its assigned

amortized cost > its actual cost

* It also allows an operation to pay for its extra actual cost using existing credit, if its assigned

amortized cost < its actual cost

* The charging method allows operations to charge cost retroactively to past:
* amortized cost of an operation = actual cost of this operation

− total cost charged to past operations

+ total cost charged by future operations

* This method deﬁnes a potential function f() that maps an algorithm conﬁguration to a value. This function f() is equivalent to the total unused credits stored up by all past operations (the bank account balance)

amortized cost of an operation = actual cost of this operation + f()

* Amortized cost can be, but does not have to be, average cost. We can assign any amortized cost to each operation, as long as they "preserve the total cost", i.e., for any sequence of operations:

amortized cost ≥ actual cost

where the sum is taken over all operations

**P vs NP**

* Some problems are hard to be solved (no fast algorithm exists)
* **Heuristic algorithms** solve hard problems approximately
  + They find a good solution, but cannot guarantee it is optimal
* Examples of heuristic algorithms:
  + Greedy algorithms – make a greedy choice at each step
  + Genetic algorithms – based on inheritance, mutation, selection, …
  + Branch and bounds – optimized backtracking with cut-offs
* P - Category in which the algorithms are solvable in polynomial time
* Polynomial -> 5N5 + 3N4 + N is O(N5)
* Not a polynomial -> 2N
* NP - Category in which the algorithms are verifiable in polynomial time.
* Decision problems
* Example:
  + Does a route shorter than L exist?
  + Yes/No

**Reductions**

* Convert problem A to problem B
  + B is a problem in P
  + Create A inputs into equivalent B inputs
  + Equivalent means that both problem A and problem B must output the same YES or NO answer for the input and converted input.
  + If B is in P then A is in P
  + If B is in NP then A is in NP
  + If A is in NP Hard then B is in NP Hard

**General Problem Solving Techniques**

* Consider you are at a computer programming exam or contest
  + You have 5 problems to solve in 6 hours
* First read carefully all problems and try to estimate how complex each of them is
  + Read the requirements, don't invent them!
* Start solving the easiest / fastest to solve problem first!
  + Leave the most complex / slow to solve problem last!
  + Approach the next problem when the previous is well tested
* Example: we are given 3 problems:
  + Cards Shuffle
    - Shuffle a deck of 52 cards in random order
  + Zig-Zag Matrix
    - Find the max-sum zig-zag path in a matrix
    - Start from the first column, go up, then down, then again up, them again down, …
  + Blocks
    - Generate all blocks 2 x 2 holding 4 of the first n Latin letters
* Read carefully the problem descriptions
  + Think a bit about their possible solutions
* Order the problems from the easiest to the most complex:
  + **Cards Shuffle**
    - Trivial – randomize the elements of array
  + **Blocks**
    - Generate variations, rotate and check for duplicates
  + **Zig-Zag Matrix**
    - Needs summing, sorting and text file processing
* Never start solving a problem without a sheet of paper + a pen
  + You need to sketch your ideas
  + Paper and pen is the best visualization tool
    - Allows your brain to think visually
* Paper works faster than keyboard / screen
* Other visualization tools could also work well
* Squared paper works best for algorithmic problems
  + Easy to draw a table
  + Easy to draw a coordinate system with objects in it
  + Easy to calculate distances
  + Easy to sketch a problem and solution ideas
* Use pens of different colors
* At the exam you have limited time!
  + Start with the problem, which will take you the least time
  + Then, again the problem, which will take you the least time
* When you achieve a result of 80/100 or 90/100
  + Think carefully for the edge cases 🡪 try to handle them
  + After you spend 10-15 minutes on the last few tests, stop!
* Don't spend hours for the last 10% of the tests!
  + Achieving a score of 80-90% of 3 problems is better than 100% of just 1 problem
* First take an example of the problem
  + Sketch it on the sheet of paper
* Next try to invent some idea that works for your example
* Check if your idea will work for other examples
  + Try to find a case that breaks your idea
  + Try challenging examples and unusual cases
* If you find your idea incorrect, try to fix it
  + Or just invent a new idea
* Consider the "cards shuffle" problem
* Idea #1: random number of times split the deck into left and right part and swap them
  + How to represent the cards?
  + How to chose a random split point?
  + How to perform the exchange?
* Idea #2: swap each card with a random card
  + How many times to repeat this?
  + Is this fast enough?
* Work decomposition is natural in engineering
  + It happens every day in the industry
  + Projects are decomposed into subprojects
* Complex problems could be decomposed into several smaller sub-problems
  + Technique known as "Divide and Conquer"
  + Small problems could easily be solved
  + Smaller sub-problems could be further decomposed as well
* Let's try idea #1:
  + Split the deck into left and right part and swap them (many times)
* Divide and conquer
  + Sub-problem #1 (single exchange) – split the deck into two random parts and exchange them
  + Sub-problem #2 – choosing a random split point
  + Sub-problem #3 – combining single exchanges
    - How many times to perform "single exchange"?
* Check-up your ideas with examples
  + It is better to find a problem before the idea is implemented
  + When the code is written, changing your ideas radically costs a lot of time and effort
* Carefully select examples for check-up
  + Examples should be simple enough to be checked by hand in a minute
  + Examples should be complex enough to cover the most general case, not just an isolated case
* What to do when you find your idea is not working in all cases?
  + Try to fix your idea
    - Sometimes a small change could fix the problem
  + Invent new idea and carefully check it
* Iterate
  + Usually your first idea is not the best
  + Invent ideas, check them, try various cases, find problems, fix them, invent better ideas, etc.
* Choose appropriate data structures before you start coding
  + Think how to represent input data
  + Think how to represent intermediate program states
  + Think how to represent the requested output
* You could find that your idea cannot be implemented efficiently
  + Or implementation will be very complex or inefficient
* Think about efficiency before writing the first line of code
  + Estimate the running time (asymptotic complexity)
  + Check the requirements
    - Will your algorithm be fast enough to conform with them?
* You don't want to implement your algorithm and find that it is slow when testing
  + You will lose time
* Best solution is sometimes just not needed
* Read carefully your problem statement
* Sometimes ugly solutions could work for your requirements and it will cost you less time
* Example: if you need to sort n numbers, any algorithm will work well for n ∈ [0..500]
* Implement complex algorithms only when the problem really needs them
* How many cards do we have?
  + In a typical deck we have 52 cards
    - No matter how fast the algorithm is – it will work fast enough
  + If we have N cards, we perform N swaps 🡪 the expected running time is O(N)
  + O(N) will work fast for millions of cards
* Conclusion:
  + The efficiency is not an issue in this problem
* Define subproblems
  + Guess part of the solution
  + Relate subprolems and solutions
  + Recurse and memoization or build DP table bottom-up
  + Check subproblems acyclic/topological order
* Solve original problem:
  + A subproblem
  + Or combination of subproblems
* Suffixes x[i:]
* Prefixes x[:i]
  + Both approaches usually run in ϴ(x)
* Substrings (subsequences) x[i…j]
  + Usually runs in ϴ(x2)
* Complexity might be exponential even for positive edges
  + Imagine graph with exponential number of edges (paths)
  + Can end up with complexity of O(2 ^ n/2)
* Will not even terminate (endless loop)
  + If there is a negative weighted cycle reachable from the source
  + If we are not careful and implement the wrong termination condition

**for v in G**

**d[v] = infinity**

**prev[v] = null**

**d[s] = 0**

**for e in G**

**relaxation step for E (u, v, w)**

**// this step is something like compare distances**

**// if you have better distance write it down etc.**

**Do the above until you can not relax any edge**

* Understand the problem
* Start from what you know
* Build a hypothesis for possible solution
  + May be incomplete or an approximate such
* Test the hypothesis
  + If it works – implement and test again
  + If it fails – analyze why did it fail?
    - If it fails for more than a single point – start again
* Complexity
  + Polynomial – Good
  + Exponential – Bad
  + Pseudo-Polynomial – Ehhh, So…So

**Exercise**

The lecture is theoretical and there are no practical problems to solve

**Evaluation & Exam**

The lecture might be helpful in choosing the best approach to solve some of the problems.