Outline

1. Reminder
2. Interval Packing
3. Greedy Algorithms
4. Algorithmic Paradigms and Complexity Theory
We have introduced the notion of graphs.
We looked at how graphs are represented in the computer.
We looked at some of the most basic algorithms: connectivity, spanning trees, and finding a path.
We introduced yet another algorithmic paradigm, brute force or backtracking.
A very important speed-up for backtracking is the “branch and bound cut”.
Algorithms and Mathematical Insight

- There are many situations where some mathematical theorems, or even simple insight produces a much faster algorithm.
- We have seen that the general packing problem is pretty tough.
- As an example, we look at the “interval packing” problem again.
- Select disjoint intervals from as set $A_i, B_i$, all lying in 1 dimensions (truck loading problem, observational problem)
Interval Packing

- Take the first interval from the left, for which the end is nearest.
- Take the next possible one, etc, until there is nothing to take.
- This algorithm is very simple and will give some kind of solution.
- But is this the maximal number disjoint intervals?

Theorem

Our algorithm is optimal. The maximum number of disjoint intervals is equal to the minimum number of points pinning the intervals.
Interval Packing: Proof

- During the algorithm, we have selected end-points $B_{i_1}, \ldots, B_{i_k}$. These “pin down” all intervals.
- Let’s assume this is not true (Reductio ad absurdum). Then there is an interval, say $A_j, B_j$ which is inside $B_{i_\nu}, \ldots, B_{i_\nu+1}$ for some $i_\nu$. Then after we selected $B_{i_\nu}, B_j$ will be closer, and it should have been selected: this contradicts our initial assumption.
- We get contradiction in the same way if the interval not pinned down is to the left of $B_{i_1}$, or to the right of the last one.
- Then any set of disjoint intervals has at most $k$ members, since each interval must contain at least one of the $B_{i_\mu}$’s selected. QED.
def intervalPack(iList):
    iList.sort(nComp)
    outList = []
    i = 0;
    while i < len(iList):
        outList.append(iList[i])
        bLast = iList[i][1]

        while i < len(iList) and iList[i][0] < bLast:
            i += 1

    return outList
def nComp(l1, l2, n=1):
    """
    compare two lists by their n-th element, default -> n = 1
    """
    if l2[n] < l1[n]:
        return 1
    elif l1[n] == l2[n]:
        return 0
    else:
        return (-1)
Cashier Problem

The cashier has at his/her disposal a collection of notes and coins of various denominations and is required to count out a specified sum using the smallest possible number of pieces.

Let us denote with $P = \{p_1, \ldots, p_n\}$ the pieces of money, each with denomination $d_i$, and the final sum with $A$.

We want to find the smallest subset $S$ of $P$ s.t.

$A = \sum_{i \in S} d_i$. 
The Objective Function

- We can represent $S$ in the computer with $n$ variables $x_i$, s.t. they take the value $x_i = 1$ iff $p_i \in S$, zero otherwise.
- We need to minimize the objective function $\sum x_i$ under the condition that $\sum x_i d_i = A$.
- How to solve this problem?
Brute Force Solution

- We have encountered Brute Force before (Backtracking): we try all possible solutions.
- We have $2^n$ possible values.
- It takes $O(n)$ to evaluate the objective function for each try.
- The running time will be $O(n2^n)$. This is exponential, i.e. extremely slow for very large $n$. 
Greedy Algorithm

- I cashier would not try all possible solutions: instead he/she would try to use the largest denominations possible first, and moving the smaller ones to count out change.
- This strategy is “greedy” (we take the largest coins first), and once selected coins are never put back.
- Is this an optimal solution? E.g. we want to count out 25 from the coins \{1, 1, 1, 1, 1, 10, 15, 20\}. The greedy strategy takes the 20 first, and counts 5 ones.
- This algorithm is suboptimal. This is not necessarily bad, especially if we have a bound on how suboptimal it is.
The Backpack Problem

We have \( N \) things, each with volume \( V_i \), and we want to pack them into backpacks of volume \( V \). There is no other constraints then the total volume: how many backpacks do we need?

- This simple problem is very general.
- E.g. you have \( N \) image processing tasks (Pan-STARRS), each of them take a certain time \( T_i \), how many computers do you need to run them, if you want the images to be processed in time \( T \)?
Greedy Solution

- Clearly, you need at least \((V_1 + \ldots + V_N)/V\) backpack.
- Optimal solution is hard, several algorithm is known, they all can be exponential (slow), like brute force.
- Greedy strategy: take the objects one by one, and put it into the first backpack, into which it fits.
- Is this optimal?
Counter Example

Consider the following objects for backpacks of $V = 6$:
- $6k$ items with volumes $1 - 2\epsilon$
- $6k$ items with volumes $2 + \epsilon$
- $6k$ items with volumes $3 + \epsilon$
- With, e.g. $\epsilon = 1/(2k)$

The optimal solution is $6k$ by taking one of each into each backpack.

The greedy algorithm will put 6 of the first object into $k$ backpacks, 2 of the second into $3k$ backpacks, and 1 of the last one into $6k$ backpacks, for a total of $10k$ (suboptimal).
Bound on Suboptimality

- Suboptimality is often not so bad, if we can give a meaningful bound.
- If $m$ is the number of backpacks found by the greedy algorithm, we can guarantee that $m < 2(V_1 + \ldots + V_N)/V$.
- Proof: we show that any pair of backpacks contain more then $V$ total volume. Let’s pick two backpacks, with number $i < j$, and total packed volume $W_i$, and $W_j$.
- Let $V_k$ denote the last object put into backpack $j$. Then $V_k \leq W_j$.
- But we could not put $V_k$ into the $i$-th backpack (otherwise the greedy strategy would have put it there), thus $V - W_i < V_k$.
Thus $W_j > V_k > V - W_i \rightarrow W_i + W_j > V$

Let’s consider the least filled backpack $i$. First, if $W_i \geq V/2$, then any other backpack has also more then $V/2$, i.e. 
\[ \sum V_s = \sum W_s \geq mV/2 \] which proves our bound.

Otherwise, if $W_i \leq V/2$, we have shown that for any $j$, $W_j > V - W_i$

\[ \sum V_s = \sum W_s > W_i + (m - 1)(V - W_i) = (m - 1)V - (m - 2)W_i \geq (m - 1)V - (m - 2)V/2 = mV/2 \]

QED.
Backpack Problem: Improvements

- It follows that $m/m_0 \leq 2$ where $m_0$ is the optimal solution.
- With more work, one can prove that the bound is approximately $\approx 1.7$. Moreover, this bound cannot be improved significantly.
- We can improve the simple greedy algorithm by sorting the items s.t. $V_1 \geq V_2 \geq \ldots V_N$.
- It can be shown that in that case $m/m_0 < 1.22\ldots$ in worst case (7 in our counter-example). The average case is similar according to numerical experiments (no theoretical results yet).
Algorithmic Patterns
By now we had examples of the following strategies:

- **Direct solution strategies:**
  - Brute force algorithms.
  - Greedy algorithms.
  - Backtracking strategies.
  - Backtracking with branch-and-bound cuts.

- **Top-down solution strategies:** Divide-and-conquer algorithms.

- **Bottom-up solution strategies:** Dynamic programming.

- **Randomized strategies:** randomization, Monte Carlo and other stochastic algorithms e.g.: simulated annealing, etc.
Tough Problems
A theoretical digression on NP-completeness

- We have seen that many problems can be solved fast with one of the above strategies. We would hope that all problems can be attacked this way.

- The bad news is that there are many important problems which are really tough, the NP-complete problems (formal definition later).

- NP-complete problems do come up surprisingly often.

- It is important to recognize an NP-complete (i.e. tough) problem, and stop banging your head into the wall trying to solve them.
What To Do?
If you encounter an NP-complete problem...

- Use a heuristic, concentrate on typical cases instead of worst case.
- Solve the problem approximately instead of exactly.
- Use an exponential time solution anyway (if you really need the exact solution).
- Choose a better abstraction. Some details ignored in the abstraction might make the difference.
- Use stochastic methods, such as simulated annealing or genetic algorithms, which will often find the solution in realistic timescales.
Computational Complexity

Some definitions

- Complexity theory classifies problems according to how tough they are.
- $P$ stands for problems which can be solved in polynomial time.
- $NP$ stands for “nondeterministic polynomial time” problem. It means that if you have the solution, you can check it quickly. These are still relatively easy: if you can guess the solution, you can test it quickly.
- $PSPACE$ problems can be solved using reasonable memory.
- $EXPTIME$ problems can be solved in exponential time.
- Undecidable: no algorithm can solve it no matter how much time or space is allowed.
The latter class might be surprising, but in fact it is the computer science equivalent of Gödel’s theorem.

An informal proof could be based on the fact that there are as many problems as there real numbers, and only as many programs as there are integers, so there are not enough programs to solve all the problems.

The above classification is coarse: it does not distinguish between $N$ or $N^{10}$ algorithms, for instance.

It does not care about the constants (for small $N$ $2^N$ can be faster than $10^6 N^3$ for instance.

Yet, these classes have practical implications, $P$ is a good approximation to problems that can be solved quickly.
Example: Simple Paths in a Graphs

- A simple path in a graph is a path without any repeated edges or vertices.
- Given a graph $G$, vertices $s$ and $t$, and a number $k$, does there exist a simple path from $s$ to $t$ with at least $k$ edges?
- This is in NP, since given a path, we can check simply (in linear time) its length.
- It’s harder to say if it is in P. In fact this is an NP-complete problem.
Example: Halting Problem

- Suppose you are working on a program, start to run it. It has been running for 5 minutes. Did it get into an infinite loop or it just takes this much time (part of the goal of this class to avoid this situation :-)

- Can a compiler tell in advance if it will run into an infinite loop? It would be very convenient...

- In fact this is a famous undecidable problem.

- I sketch out the proof. Note that this is roughly equivalent to Gödel’s theorem.

- According to some, this shows that people are smarter than computers (there are problems computers cannot solve).
The Universal (Turing) Machine

- According to Turing we imagine an ideal computer, which i) never makes mistakes, ii) can work up to arbitrary time iii) its memory can be extended arbitrarily.
- Mathematical definition of an algorithm: a procedure which can be programmed on an ideal computer.
- A universal machine $T$ is one on which any algorithm is programmable. In particular, any realistic computer can be simulated on a universal machine.
The Universal (Turing) Machine

- $T$ has three slots for “tapes”: we can input the program and the data on the first two, and when it finished, it prints the output on the third.

- The code $\bar{P}$ of program $P$ is a series of 0’s and 1’s on a tape. We can represent the results of a program $P$ on data $x$ as $y = T(P, x)$, where $x$ and $y$ are data vectors of 0’s and 1’s.
Halting Problem on a Turing machine

- Given $P$ and $x$, decide if it will work for ever, or will it stop in finite time?
- Look at the special case of $x = \bar{P}$, where $\bar{P}$ represents the code of the program $P$.
- If an algorithm exists to decide the above problem, we can code it on the Turing machine. Let’s write the program $Q$ such a way that it will stop if answer is yes, i.e. the program $P$ would run for infinite time. Otherwise let’s program it to go into an infinite loop (that’s the trick).
Let’s apply $Q$ on itself.

If $Q$ stops, then $Q$ algorithm applied to $\bar{Q}$ is infinite. But that’s a contradiction!

Similarly, there is a contradiction if $Q$ enters into in infinite loop.

Therefore algorithm $Q$ cannot exist and we proved the following:

**Theorem**

*The Halting Problem is undecidable.*
The most famous open problem in theoretical science is whether $P = NP$. (beware that here $P$ is not a program!)

In other words, if it’s always easy to check a solution, should it also be easy to find the solution?

Most theorists think that there is no reason to be so, but there is no proof.

There is no known problem in NP, which is proven to be not in P.
Problem A is easier than B, we write $A < B$ if A can be solved by an algorithm with a small number of calls to B.

There are some variants of the definition depending on the meaning of *small*.

A problem in A in NP is **NP-complete**, if for any other problem $B$ in NP $B < A$.

This is a strong definition, it seems to imply that the two problems have to be closely related.
Cook’s Theorem

**Theorem**

An NP-complete problem exist.

- Bounded halting. Input a program X and a number K. The problem is to find data which, when given as input to X, causes it to stop in at most K steps.
- This is in NP, since given a data, we can simulate the program and let it run for K steps. (We would need to be careful to define steps, programs etc).
- Given any problem A in NP, we can write a program, which tests in polynomial time $p(n)$ any given solution.
Cook’s Theorem Cont’d

- If we modify the above program to go into infinite loop, whenever the given data set is not a solution, we can pass it to the bounded halting problem.
- I.e. we “proved” (lots of details omitted) that arbitrary \( A < \) bounded halting, i.e. the latter is an NP-complete problem.
- This is a pretty abstract existence proof.
- In practice NP-completeness is proven based on the observation that if \( A < B \) and \( B < C \) then \( A < C \)
- I.e. if \( A \) is NP-complete, and \( A < B \), then \( B \) is also NP-complete.
What is the point of all this? NP-complete problems are in a precise sense the toughest problems in NP. Even though we don’t know if there are any problems in NP that are not in P, but if we think that $NP \neq P$, NP-complete problems will not have an easy solution.

Conversely, if $NP = P$ were true, we could find polynomial solutions to all such problems.
Summary

- We have demonstrated with the packing problem how to obtain fast solutions with mathematical insight and theorems.
- We looked at greedy algorithms.
- We summarized algorithmic patterns.
- We looked at complexity theory.
Homework # 8

(E1) Variation on interval packing: a truck is going through the route, as in the lecture, but wants to do all the possible jobs. Design a simple algorithm, which will do it with the minimum number of roundtrips. What determines the minimum number of trips? (Hint: look at how many intervals are pinned down by arbitrary points along the route).

(E2) Implement the brute force and greedy algorithms for the counting change. Fix the cashiers inventory with 200 bills. Create a simulation, where you run both the brute force and greedy algorithm on each possible sums which can be paid with the given bills. Plot and discuss the results.
Homework # 8
continued

(E3) Implement the backpack problem with sorting. Simulate any meaningful application you could think of, and try to verify numerically by comparing to the brute force solution the bound quoted in the lecture.