29 June

Introduction to computational complexity [P&S Chap. 8]

Problems vs. instances [see P&S §1.2]

Up to this point in the course, I have been somewhat loose in my usage of the word "problem." But there is a distinction between a problem and an instance of a problem, and it is important to keep these two ideas separate, so from now on I will be using these words carefully. (Except problem sets will still be called problem sets, even though perhaps they should be called "instance sets.")

- A problem is described by specifying, in general terms, the form of the input and the desired properties of the output.

- An instance of a problem is a specific input (which can consist of numerical data, a graph, sets of linear inequalities, etc.).

Examples.

- **PROBLEM:** Integer multiplication.
  Input: Two integers m and n.
  Output: The product mn.

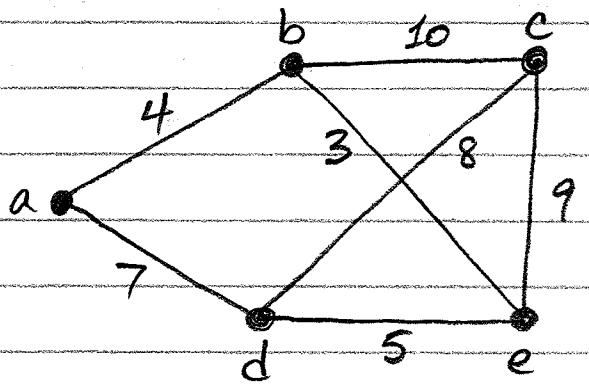
- **INSTANCE:** 341×87.

- **INSTANCE:** -6×408.

- **INSTANCE:** 15208×68818.

- **PROBLEM:** Minimum spanning tree.
  Input: A connected undirected graph $G=(V,E)$ and a function $w: E \to \mathbb{R}$ specifying edge weights.
  Output: A spanning tree (V,T) of minimum total edge weight.

- **INSTANCE:**



29 June

Time bounds and asymptotic notation [P&S §8.2]

Question: How much time will an algorithm take to solve an instance of a problem?

Observation: Different computers run at different speeds, so it is not meaningful to answer this question in terms of seconds. Instead, when we say "time" we will really mean the number of elementary operations that must be performed.

- "Elementary operations" can include:
  - Basic arithmetic operations (addition, subtraction, multiplication, division, remainder)
  - Comparisons (e.g., evaluation of $\leq$ or $=$ to see whether it's true or false)
  - Reading a value from memory, writing a value to memory, swapping the values of two variables
  - Branching, looping, function call and return
- Often in a particular algorithm one kind of elementary operation dominates. For example, in a sorting algorithm, most of the elementary operations are comparisons. So, to make the analysis

simpler, we count the number of elementary operations of the dominant kind.

The "running time" of an algorithm on an instance is the number of elementary operations performed by the algorithm to solve the instance.

Example. Finding the minimum element in an unordered list. (This is the problem.)

Input: A nonempty (finite) list of n integers.

Output: The minimum integer that appears in the list.

Instance: $(6, 3, 10, -4, 8, -7, 2.)$ [so $n=7$]

Algorithm:

1. Set $m :=$ (first element of list), $i := 2$.
2. If $i > n$, return m (and stop).
3. Let p be the ith element of the list. If $p < m$, set $m := p$.
4. Set $i := i+1$ and go back to step 2.

29 June

How many elementary operations does this algorithm take to solve the given instance?

	Elementary ops
1. Set $m := 6$, $i := 2$.	2 assignments
2. If $i > 7$... — false.	1 comparison
3. If $3 < 6$ (true), set $m := 3$.	1 comp., 1 assign
4. Set $i := 2+1$, go back to step 2.	1 add, 1 assign, 1 jump
2. If $i > 7$... — false.	1 comparison
3. If $10 < 3$... — false.	1 comparison
4. Set $i := 3+1$, go back to step 2.	1 add, 1 assign, 1 jump
2. If $i > 7$... — false.	1 comparison
3. If $-4 < 3$ (true), set $m := -4$.	1 comp., 1 assign
4. Set $i := 4+1$, go back to step 2.	1 add, 1 assign, 1 jump
2. If $i > 7$... — false.	1 comparison
3. If $8 < -4$... — false.	1 comparison
4. Set $i := 5+1$, go back to step 2.	1 add, 1 assign, 1 jump
2. If $i > 7$... — false.	1 comparison
3. If $-7 < -4$ (true), set $m := -7$.	1 comp., 1 assign
4. Set $i := 6+1$, go back to step 2.	1 add, 1 assign, 1 jump
2. If $i > 7$... — false.	1 comparison
3. If $2 < -4$... — false.	1 comparison
4. Set $i := 7+1$, go back to step 2.	1 add, 1 assign, 1 jump
2. If $i > 7$ (true), return -7.	1 comp., 1 return

TOTAL: 37 elementary operations.

Normally we don't analyze algorithms in this much exacting detail because it's way too tedious.

Instead, we simplify the analysis (while also making it more general and hence more useful):

- We consider all instances of size n and find an upper bound on the number of elementary operations in the worst case (as a function of n).

- We ignore constant coefficients and lower-order terms in this function by using asymptotic notation.

  Justification:
  - The difference between an algorithm that takes n steps and an algorithm that takes $2n$ steps vanishes if the second is run on a computer that is twice as fast as the first.
  - But for large enough n, an algorithm that takes n steps will always be faster than an algorithm that takes n^2 steps, regardless of the difference in speed between the two computers.

29 June

Asymptotic notation [P&S Defn 8.1]

Defn. Let f and g be functions from the positive integers to the positive reals.

(a) We write $f(n) = O(g(n))$ if there exist constants $c > 0$ and $N \geq 1$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.

(b) We write $f(n) = \Omega(g(n))$ if there exist constants $c > 0$ and $N \geq 1$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.

(c) We write $f(n) = \Theta(g(n))$ if there exist constants $c, c' > 0$ and $N \geq 1$ such that $c \cdot g(n) \leq f(n) \leq c' \cdot g(n)$ for all $n \geq N$.

Observe:

- $f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$.

  $[\ f(n) \geq c \cdot g(n) \iff g(n) \leq \frac{1}{c} \cdot f(n)\ ]$

- $f(n) = \Theta(g(n))$ iff $[f(n) = O(g(n))$ and $f(n) = \Omega(g(n))]$.

Intuition:

- $f(n) = O(g(n))$ means that $f(n)$ is asymptotically less than (some constant multiple of) $g(n)$.
- $f(n) = \Omega(g(n))$ means that $f(n)$ is asymptotically greater than (some constant multiple of) $g(n)$.

Note: $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$ can be viewed as sets of functions $f(n)$ that satisfy the definition, in which case we should write $f(n) \in O(g(n))$, etc. But it is common practice to write $f(n) = O(g(n))$.

Be careful, though! This isn't really equality: $f(n) = O(g(n))$ and $h(n) = O(g(n))$ do not imply $f(n) = h(n)$.

29 June

Alternative (equivalent) definitions

Defn Let f and g be functions from the positive integers to the positive reals.

(a) We write $f(n) = O(g(n))$ if

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} \overset{\text{def}}{=\!=} \inf_{n\in\mathbb{N}} \sup_{k\geq n} \frac{f(k)}{g(k)} < \infty.$$

(b) We write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.

(c) We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Note: If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then $\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f(n)}{g(n)}$.

So if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists (and is less than ∞), then $f(n) = O(g(n))$. But $\limsup_{n\to\infty} \frac{f(n)}{g(n)}$ always exists, even if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not [though possibly $\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \infty$], so the converse is not necessarily true.