30 June

Asymptotic notation

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

$f(n) = O(g(n))$ means:

(Equivalent definitions.)

- $\exists\ c>0,\ N \geq 1\ \forall\ n \geq N\ [f(n) \leq c \cdot g(n)]$
- $\limsup_{n\to\infty} f(n)/g(n) < \infty$

} $g(n)$ is an asymptotic upper bound for $f(n)$

$f(n) = \Omega(g(n))$ means:

- $\exists\ c>0,\ N \geq 1\ \forall\ n \geq N\ [f(n) \geq c \cdot g(n)]$
- $g(n) = O(f(n))$
- $\limsup_{n\to\infty} g(n)/f(n) < \infty$

} $g(n)$ is an asymptotic lower bound for $f(n)$

$f(n) = \Theta(g(n))$ means:

- $\exists\ c, c' > 0,\ N \geq 1\ \forall\ n \geq N\ [c \cdot g(n) \leq f(n) \leq c' \cdot g(n)]$
- $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $\liminf_{n\to\infty} f(n)/g(n) > 0$ and $\limsup_{n\to\infty} f(n)/g(n) < \infty$

(New)

$f(n) = o(g(n))$ means:

- $\forall\ \varepsilon > 0\ \exists\ N \geq 1\ \forall\ n \geq N\ [f(n) \leq \varepsilon \cdot g(n)]$
- ~~$f(n) = O(g(n))$ but $g(n) \neq O(f(n))$~~ ← [This isn't quite right.]
- $\lim_{n\to\infty} f(n)/g(n) = 0$

$f(n) = \omega(g(n))$ means:

- $\forall\ c > 0\ \exists\ N \geq 1\ \forall\ n \geq N\ [f(n) \geq c \cdot g(n)]$
- $g(n) = o(f(n))$
- $\lim_{n\to\infty} f(n)/g(n) = \infty$

Notation: We write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$.

Example: Some common functions.

$$1 \ll \log\log n \ll \log n \ll (\log n)^2$$

Polynomial and subpolynomial

$$\ll n^{1/3} \ll n^{1/2} \ll n \ll n \log n$$

$$\uparrow \quad \ll n^2 \ll n^3 \ll n^4 \ll n^5 \ll n^6 \ll \cdots$$

Superpolynomial ↓

$$\ll 2^n \ll 3^n \ll 2^{n^2} \ll n! \ll n^n \ll 2^{2^n} \ll \cdots$$

Example. Constant coefficients don't matter.

$5n^2 = O(n^2)$ because $\lim_{n\to\infty} \frac{5n^2}{n^2} = 5 < \infty$.

Example. Lower-order terms don't matter.

$n^3 + 4n + \log n = O(n^3)$ because

$$\lim_{n\to\infty} \frac{n^3 + 4n + \log n}{n^3} = 1 < \infty.$$

30 June

The size of an instance [P&S §8.3]

The size of an instance is the length of an encoding of the input.

- Input is presented to a computer program as a string of symbols from some alphabet: bits, typewriter symbols, Unicode characters, etc.

Example. Integer factorization.

- The input is a single positive integer.

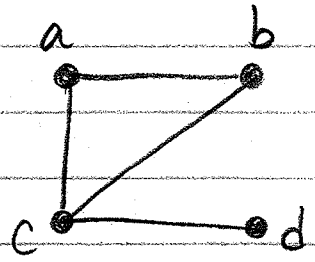
- The size of the instance is not the value of the integer but the number of digits in a representation of the integer in some base b, which is $\lceil \log_b (n+1) \rceil$.

- Regardless of what base is used, the size of the instance is $\Theta(\log n)$, because $\log_b n = \log n / \log b$ and b is a constant.

Example. Minimum spanning tree.

- The input is a graph with edge weights.
  - Say $|V| = n$, $|E| = m$.
- We can represent a graph by its adjacency matrix:

	a	b	c	d
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0



This requires n^2 bits.

- For a simple undirected graph, we can improve this representation somewhat by only giving the upper triangular part of the adjacency matrix (because the matrix is symmetric), not including the diagonal (which will always be zeroes). This requires $\binom{n}{2}$ bits, which is less than n^2, but $\binom{n}{2} = \Theta(n^2)$ so we haven't gained anything asymptotically.

- If the graph is sparse ($m \ll n^2$), then we can improve the size by using adjacency lists: for each vertex, list its neighbors. This is a total of $2m$ integers (representing each vertex as an integer).

30 June

Size of an instance — (2)

— Each of these integers can be represented by $\Theta(\log n)$ bits, so the size of this representation is $\Theta(m \log n)$.

— In practice, computers use a fixed amount of space for all integers within their range (e.g., 0 to $2^{32}-1$). This fixed amount of space is called a <u>word</u>, and every integer within range occupies exactly one word. So often we don't worry about counting the bits within each integer — we just count the number of integers. Hence, we say that the size of an adjacency-list representation of a graph is $\Theta(m)$.

— If $m \ll n^2$, then this is an asymptotic improvement.
  — e.g., trees, planar graphs, graphs with bounded degree, etc.

— Note: Input for integer factorization is typically bigger than will fit in a single word, so there we <u>do</u> need to count bits.

— List of edge weights is $\Theta(m)$ more integers, so total size of an instance of the minimum spanning tree problem is $\Theta(m)$.

— In the analysis of graph algorithms, it is common to see both n and m used to describe sizes of instances or running times of algorithms.

30 June

Analysis of algorithms [P&S §8.4]

Example. What is the running time of Dijkstra's algorithm to find shortest distances from node s in a graph with n nodes and m arcs?

DIJKSTRA (arc weight c_{si})

1. Initialize $W := \{s\}$, $\rho(s) := 0$, $\rho(i) = c_{si}$ for $i \neq s$.
2. If $W = V$ (all vertices are in W), we are done.
3. Find $\min\{\rho(y) : y \notin W\}$, say $\rho(x)$.
4. Add x to W.
5. For all $y \notin W$, set $\rho(y) := \min\{\rho(y), \rho(x) + c_{xy}\}$.
6. Go to step 2.

Step 1 is executed once, and requires $\Theta(n)$ elementary operations.

Steps 2 through 6 are executed $n-1 = \Theta(n)$ times. On each one of these iterations:

- Step 2 can be done in constant time, i.e., $O(1)$ steps, by comparing the cardinalities of W and V (assuming that the data structure being used for these sets keeps track of this information).
- Step 3 can be done in $O(n)$ elementary operations (e.g., algorithm from yesterday for finding the minimum element in

a list of integers).
- Step 4 can be done in a constant number of elementary operations.
- Step 5 takes $O(n)$ iterations, each of which takes constant time, so step 5 overall takes $O(n)$ time.
- Step 6 is a single elementary operation.

So a single iteration of steps 2–6 together takes $O(n)$ elementary operations. There will be $\Theta(n)$ such iterations, so in all these iterations take $O(n^2)$ time.

Therefore, overall Dijkstra's algorithm takes $O(n^2)$ time.

← This is the time complexity of this algorithm.

30 June

Polynomial-time algorithms [P&S §8.5]

Defn. An algorithm runs in polynomial time if its worst-case running time on an instance of size n is $O(g(n))$ for some polynomial $g(n)$.

Example. Dijkstra's algorithm runs in $O(n^2)$ time, and n^2 is a polynomial, so this is a polynomial-time algorithm.

The distinction between polynomial-time algorithms and superpolynomial-time algorithms is considered the boundary between (theoretically) "efficient" algorithms and inefficient algorithms.

Advantages of polynomial-time algorithms:

— "Polynomial-time" is robust against changes in the model of computation. Given any two "reasonable" models of computation, any algorithm that runs in polynomial time in one model can also be made to run in polynomial time in the other, with at most a polynomial slowdown [e.g., $O(n^5)$ time instead of $O(n^2)$].

— A polynomial-time algorithm can be viewed as a subroutine that can be called by other algorithms as part of solving larger problems. As long as the subroutine is called only polynomially many times, the total number of elementary operations will still be polynomial.

— Experience shows that once a polynomial-time algorithm is discovered for a problem, further research can often reduce the exponent and lead to a practically useful algorithm. Maybe the original algorithm is $\Theta(n^{10})$, for instance, but with improvements this can be brought down to $O(n^3)$. The hard part is often getting a polynomial-time algorithm in the first place.

— A well known example of reducing the exponent: Matrix multiplication. The straightforward algorithm for multiplying two $n \times n$ matrices takes $\Theta(n^3)$ steps. In 1969 Strassen invented an algorithm to do it in $O(n^{\log_2 7}) = O(n^{2.8074})$ steps. Current best known exponent is about 2.3729; the optimal value is an open question.