

6.1 Overview of next few lectures

- Computational Problems
- Models of Computation
- Turing Machine and the Church-Turing Thesis
- Complexity Classes

6.2 Motivating Story

Recall the Minimum Spanning Tree (MST) problem from Lectures 2 and 3. Now consider a closely related problem called the Minimum Steiner Tree problem (or simply Steiner).

Minimum Steiner Tree;

INPUT: Weighted Graph $G = (V, E)$ with weights $w : E \rightarrow \mathbf{R}^{\geq 0}$, and a set of *terminals* $S \subset V$.

OUTPUT: A tree $T \subseteq E$ of minimum weight $w(T) = \sum_{e \in T} w(e)$ such T connects all vertices in S .

Question: How to MSpanningT and Steiner compare with each other? Which is an “easier” problem?

Turns out Steiner seems pretty hard. While we know how to solve MST in nearly linear time ($O(n \log n + m)$ etc.), we don't know how to solve Steiner in time less than 2^n .

Moral: Complexity is subtle and quick to change! (with apologies to Tolkien “Do not meddle in the affair of wizards, for they are subtle and quick to anger.”)

Here's another pair of problems:

Shortest Path;

INPUT: Weighted Graph $G = (V, E)$ with weights $w : E \rightarrow \mathbf{R}^{\geq 0}$ and a pair of vertices s, t .

OUTPUT: The shortest path from s to t , i.e., sequence of edges e_1, \dots, e_k minimizing $\sum_{i=1}^k w(e_i)$ where e_1 is incident to s , and e_k to t and e_i and e_{i+1} meet at a vertex for every i .

One can define **Longest Path** similarly (while stressing that a *path* may not visit the same vertex twice, or else there can be infinitely long paths in graphs).

Question: How do Shortest Path and Longest Path compare with each other?

Question: How do Steiner and Shortest Path compare? Etc.?

More general questions: Why can't we solve every problem in time $O(n)$? Or can we (and we are just holding back)?

Can we solve every problem in time 2^n ?

How about 2^{2^n} - surely that is sufficient?

In order to study such questions, we first need to make the notion of an algorithm and its complexity a bit more precise. But even more basic: We need to start with defining what a computational problem is!

6.3 Computational Problems

Usually we think of a computational problem as a *function* which tells us what the output should be for a given input. We often represent inputs and outputs by a finite sequence of bits. Such finite sequences are denoted $\{0, 1\}^* = \cup_{n \geq 0} \{0, 1\}^n$.

But functions don't capture all problems naturally. E.g., in MST given an input graph we don't insist on a particular minimum spanning tree - any MST will do. So the right mathematical structure that captures computational problems are *relations*.

(Recall basic math: A relation R between the input set $I = \{0, 1\}^*$ and output set $O = \{0, 1\}^*$ is simply a subset $R \subseteq I \times O$. A relation R is a function if for every $x \in I$ there exists a unique $y \in O$ such that $(x, y) \in R$. For our purposes it is better to not make the restrictions that every input should have exactly one valid output.)

Question: When does a graph not have an MST?

Even though computational problems are naturally described by functions and relations, we will actually try to work with a much more basic concept, namely a set, aka *language*.

A language $L \subseteq I$ represents the computational problem: Given $x \in I$, determine if $x \in L$?

Example: $L_{\text{Steiner}} = \{(G, S, k) \text{ s.t. } G \text{ has a Steiner tree of cost } \leq k \text{ connecting } S\}$.

Claim: Can solve L_{Steiner} if we can solve Steiner. But converse is also true. Why?

Why Languages? You could mean two things by that question. (1) Why call sets languages? Its just linguistics!
(2) Why work with languages: They manage to capture the essence of most computational problems, but are also easy (or at least easier to) relate complexity of different problems.

So while it would be hard to compare Steiner and Longest Path directly, it is possible to compare L_{Steiner} and a language $L_{\text{LongestPath}}$ (that captures Longest Path) more naturally. In particular there exists an algorithm A running in polynomial time that takes as input (G, S, k) and produces an output (H, L) such that $(G, S, k) \in L_{\text{Steiner}}$ if and only if $(H, L) \in L_{\text{LongestPath}}$. Here, as you might imagine, we define $L_{\text{LongestPath}}$ as $\{(H, L) \text{ s.t. } H \text{ has a path of length at least } L\}$.

Question: What does the existence of A tell you about Steiner tree vs. Longest Path?

(We will talk about such algorithms later, and while we won't show one to you, it will be implied by statements we show!)

6.4 Models of Computation

What? - Mathematical model specifying a computer.

Why? - Would like to measure more precisely what a computer can or can not do in a given number of steps (or with a given amount of memory). Needs a precise model of number of steps. Model should capture any computer we can build, to within constant, or small factors.

In what follows we will describe three models of computing: (1) Circuits, (2) Word-RAM and (3) Turing Machine.

Of the three, the TM will appear least natural, so we should understand why we will dismiss the more natural models and go to the least natural one for the rest of the course!

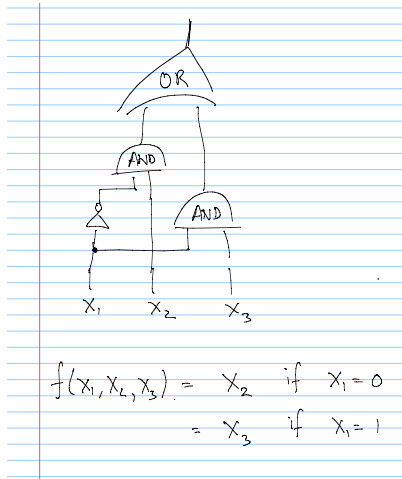


Figure 6.1: Figure of Circuit

6.5 Circuits as Models of Computing

See Figure 6.5 for an example circuit. Circuits form a model of computing, once you fix the input length. Size of a circuit, measured in number of gates is a measure of complexity.

Formally a circuit computing an m bit function of n bit inputs, over gate set $\{\text{AND}, \text{OR}, \text{NOT}\}$ is given by a directed acyclic graph with labels on vertices and some additional information as follows: There are n “input vertices” labelled x_1, \dots, x_n - these vertices have no edges pointing in to them. There are m output vertices labelled y_1, \dots, y_m which have exactly one edge pointing into them. All other vertices are labelled by AND, OR, or NOT. NOT gates have exactly one edge going in to them and AND and OR gates have two edges pointing into them. The figure and example show how a circuit computes a function.

Example 1: Addition Circuit: $2n$ inputs x_1, \dots, x_n and y_1, \dots, y_n and $n + 1$ outputs z_1, \dots, z_{n+1} .

Addition circuit (see Figure 6.5) chains $n + 1$ units of the same type with the input to the i th unit being c_i, x_i and y_i and outputs being z_{i+1} and c_{i+1} with the feature that $z_{i+1} + 2c_{i+1} = c_i + x_i + y_i$. We won't describe the details of this “3-bit” adder and it is not important. It suffices to note that a 3-bit adder of constant size exists. See following proposition.

History of Circuits: Introduced as formal model of computing by Shannon [S38] (as his Master's thesis at MIT). Shannon came up with the model and then realized it was built on Boolean logic [B47].

Proposition 6.1. *AND, OR, NOT form a complete basis for Boolean functions. Specifically for every k and every function $f : \{0, 1\}^k \rightarrow \{0, 1\}$ there is a circuit with $O(2^k)$ gates that computes f .*

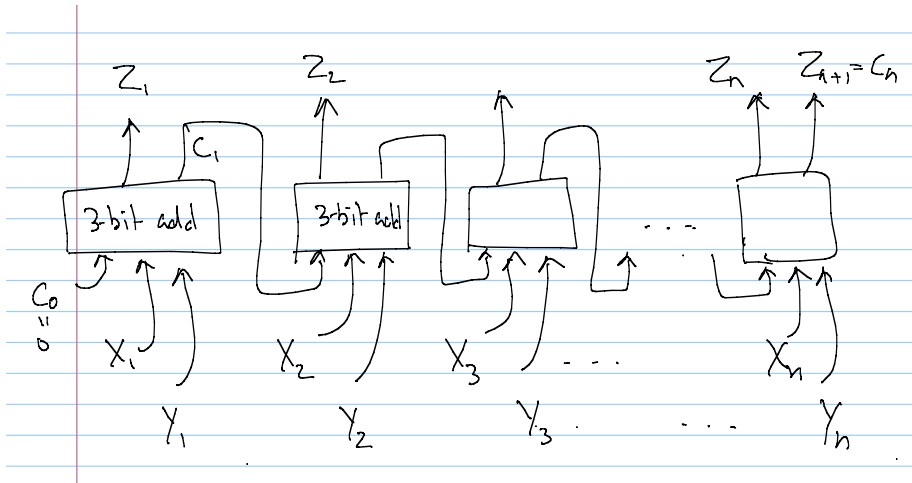


Figure 6.2: Figure of n-bit adder

Proof. Hardwire the truth table for f and select the right output. □

In fact, the optimal bound is $O(2^k/k)$, but we will not cover this in class. More specifically, every $f : \{0, 1\}^k \rightarrow \{0, 1\}$ can be implemented by a circuit of this size, and for every k in fact *most* functions require a circuit of size $\Omega(2^k/k)$. This latter fact was proven by Shannon and is proven via a counting argument.

Basic facts about circuits:

1. Choice of basis only affects circuit size by constant factor. (Why?)
2. Smallest circuit with a fixed basis is well-defined. (But can be hard to find!)
3. Most functions require exponential size.
4. Every function has exponential size (see Proposition above). Can't distinguish complexity of finding the circuit vs. implementing the circuit.
5. Circuit size never smaller than n — can't distinguish description complexity from running time complexity.

Because of the last two “cons” we prefer a non-hardware oriented model of computing. The next model is closest to our notion of an algorithm (though a bit more restrictive so we can write down the full specs in bounded time and space!)

6.6 A RAM Model

We will start by formalizing the model that we've implicitly been using so far, and is commonly used for the analysis of algorithms: the Word RAM Model (where RAM stands for Random-Access Memory). At a high level, its features are as follows:

- Main memory is an array $M = (M[0], \dots, M[S-1])$ of w -bit *words*, which are also interpreted as numbers in the range $\{0, \dots, 2^w - 1\}$
- We also have an array $R = (R[0], \dots, R[r-1])$ of $r = O(1)$ *registers*, which are also w -bit words.
- Algorithms only have direct access to the registers. The remaining memory locations can only be accessed by indirect addressing: interpreting a word (in a register) as a pointer to another memory location. So we require $S \leq 2^w$.
- A program is a finite sequence of instructions.
- The instruction set consists of basic arithmetic and bitwise operations on words (in registers), conditionals (if-then), goto, copying words between registers and main memory, halt.
- We also allow *malloc* — add an additional memory cell, incrementing S (and w if needed to ensure $S \leq 2^w$).
- The initial memory configuration is as follows:
 - The input of length n is placed in main memory locations $(M[0], \dots, M[n-1])$.
 - The word size is set to be $w = \lceil \log_2 \max\{n+1, k+1\} \rceil$, where k is an upper bound on the numbers occurring in either the input or as a constant in the program.
 - Register 0 is initialized with w and register 1 with n .

Formally, we allow the following instructions, for any $i, j, k, m \in \mathbf{N}$,¹:

- $R[i] \leftarrow m$
- $R[i] \leftarrow R[j] + R[k]$
- $R[i] \leftarrow R[j] - R[k]$
- $R[i] \leftarrow R[j] \cdot R[k]$

¹For computer scientists, the natural numbers \mathbf{N} include zero!

- $R[i] \leftarrow R[j] \wedge R[k]$ (bitwise AND)
- $R[i] \leftarrow \neg R[j]$ (bitwise negation)
- $R[i] \leftarrow \lfloor R[j]/R[k] \rfloor$
- $R[i] \leftarrow M[R[j]]$
- $M[R[j]] \leftarrow R[i]$
- IF $R[i] = 0$, GOTO ℓ .
- HALT
- MALLOC

Q: What about left-shift, right-shift, mod, XOR, $>$, max?

Definition 6.2. A *word-RAM program* is any finite sequence of instructions $P = (P_1, P_2, \dots, P_q)$ of the types listed above. The number r of registers is implicitly defined to be the largest direct memory address (the numbers i, j, k in the instruction set) occurring in the program.

6.7 Formalizing Computation

The notion of a *configuration* is meant to capture the entire state of a computation at a point in time — everything that is needed to determine future behavior.

Definition 6.3. A *configuration* of a word-RAM program P is a tuple $C = (\ell, S, w, R, M)$, where $\ell \in \{1, \dots, q+1\}$ is the *program counter*, $S \in \mathbf{N}$ is the *space usage*, $w \in \mathbf{N}$ is the *word size*, $R = (R[0], \dots, R[r-1]) \in \{0, \dots, 2^w - 1\}^r$ is the *register array*, and $M = (M[0], \dots, M[S-1]) \in \{0, \dots, 2^w - 1\}^S$ is the *memory array*.

Definition 6.4. For a configuration $C = (\ell, S, w, R, M)$ of a word-RAM program $P = (P_1, P_2, \dots, P_q)$, we define the *next configuration* $C' = (\ell', S', w', R', M')$, writing $C \Rightarrow_P C'$, as follows:

- if $P_\ell = \text{“IF } R[i] = 0, \text{ GOTO } m\text{”}$ for some $i < r$ and $m \in \{1, \dots, q\}$, then $C' = (\ell', S, w, R, M)$ where $\ell' = m$ if $R[i] = 0$ and $\ell' = \ell + 1$ otherwise.

- if $P_\ell = "R[i] \leftarrow m"$ for some $i < r$, then $C' = (\ell + 1, S, w, R', M)$ where $R'[i] = m \bmod 2^w$ and $R'[j] = R[j]$ for all $j \neq i$.
- if $P_\ell = "R[i] \leftarrow R[j] + R[k]"$ for some $i, j, k < r$, then $C' = (\ell + 1, S, w, R', M)$ where $R'[i] = (R[j] + R[k]) \bmod 2^w$ and $R'[m] = R[m]$ for all $m \neq i$.
- if $P_\ell = "M[R[j]] \leftarrow R[i]"$ where $i, j < r$ and $R[j] < S$, then $C' = (\ell + 1, S, w, R, M')$ where $M'[R[j]] = R[i]$ and $M'[k] = M[k]$ for all $k \neq R[j]$.
- \vdots
- if $P_\ell = "MALLOC"$, then $C' = (\ell + 1, S + 1, w', R, M')$, where $w' = \max\{w, \lceil \log_2(S + 1) \rceil\}$, $M'[S] = 0$ and $M'[i] = M[i]$ for $i = 0, \dots, S - 1$.
- Otherwise (in particular if $P_\ell = \text{HALT}$ or $\ell = q + 1$), $C' = (q + 1, S, w, R, M)$.

Now we need to define what it means for a program to solve a problem — actually first we need to say what we mean by a “problem”.

Definition 6.5. A *computational problem* is a function $f : \mathbf{N}^* \rightarrow 2^{\mathbf{N}^*}$.² $f(x)$ is the *set of correct answers* on input x .

Definition 6.6. A RAM program $P = (P_1 \dots, P_q)$ with $r \geq 2$ registers *solves* the problem $f : \mathbf{N}^* \rightarrow 2^{\mathbf{N}^*}$ if for every input $x = (x_1, \dots, x_n) \in \mathbf{N}^*$ and $k \geq \max\{x_1, \dots, x_n, m\}$ where m is the largest constant occurring in P , there is a sequence of configurations C_0, \dots, C_t such that:

- $C_0 = (1, n, w, R, M)$ for $w = \lceil \log_2(\max\{n, k + 1\}) \rceil$, $R[0] = n$, $R[1] = k$, $R[2] = R[3] = \dots = R[r - 1] = 0$, $M[0] = x_1$, $M[1] = x_2, \dots, M[n - 1] = x_n$,
- $C_{i-1} \Rightarrow_P C_i$ for all $i = 0, \dots, t$,
- $C_t = (q + 1, S, w, R, M)$ where $(M[1], M[2], \dots, M[R[0] - 1]) \in f(x)$.

Q: Why don't we just look at computing functions $f : \mathbf{N}^* \rightarrow \mathbf{N}^*$?

Some key points:

² \mathbf{N}^* denotes sequences of 0 or more natural numbers. $2^{\mathbf{N}^*}$ denotes the powerset of \mathbf{N}^* , also written $P(\mathbf{N}^*)$.

- Uniformity: we require that there is a single finite program should work for arbitrary inputs of unbounded size (as $n, k \rightarrow \infty$).
- Computation proceeds by a sequence of “simple” operations.
- We do not impose an a priori bound on time (# steps) or space (memory). These are *resources* that we will want to minimize, but we still consider something an algorithm even if it uses huge amounts of time and space.

6.8 Examples

On the next couple of pages are Word-RAM implementations of Counting Sort and Merge Sort. As you can see, it is quite tedious to write out all the details of our algorithms in the Word-RAM model (it is like programming in Assembly Language), so we will only do it this once. The point is to convince ourselves that the Word-RAM model really is sufficient to implement the algorithms we’ve studied, with the asymptotic efficiency that we’ve claimed. We also want to become aware of some subtleties of the model (to be discussed later) that we should be careful about when working with higher-level descriptions of algorithms.

You may note that the implementations of CountingSort and MergeSort will have some steps that are not strictly within our instruction set, but they all can be implemented using a constant number of actual instructions (possibly with a constant number of additional registers), so this is only for notational convenience.

Q: how can we implement recursion (like in MergeSort) on the Word-RAM model, which has no procedure calls in its instruction set?

Algorithm 1: CountingSort

```

/* input in  $M[0], \dots, M[R[0]-1]$ . */
/* tallies will be kept in  $M[R[0]], \dots, M[R[0]+R[1]]$  */
/* sorted array will be first be placed in  $M[R[0]+R[1]+1], \dots, M[2R[0]+R[1]]$ ,
   and copied at end to  $M[0], \dots, M[R[0]-1]$  */
/* throughout we use  $R[2]$  as a loop counter */
/* loop to allocate  $R[1]+1$  memory cells for tallies */
1 MALLOC;  $R[2] \leftarrow R[1]+1$ 
2 IF  $R[2] = 0$ , GOTO 6
3 MALLOC
4  $R[2] \leftarrow R[2]-1$ 
5 GOTO 2

/* loop to compute tallies, and allocate memory for sorted list */
/*  $M[R[0]+i]$  will contain tally for items of type  $i$  */
6 IF  $R[2] = R[0]$ , GOTO 11
7  $M[R[0]+M[R[2]]] \leftarrow M[R[0]+M[R[2]]]+1$ 
8  $R[2] \leftarrow R[2]+1$ 
9 MALLOC
10 GOTO 6

/* loop to compute prefix sums of tallies */
/*  $M[R[0]+i]$  will contain number of items of type less than  $i$  */
/*  $R[3]$  is running tally */
11  $R[2] \leftarrow R[0]; R[3] \leftarrow 0$ 
12  $R[4] \leftarrow M[R[2]]$ 
13  $M[R[2]] \leftarrow R[3]$ 
14  $R[3] \leftarrow R[3]+R[4]$ 
15 IF  $R[2] = R[1]+R[0]$  GOTO 18
16  $R[2] \leftarrow R[2]+1$ 
17 GOTO 12

/* Loop to compute sorted array in locations  $M[R[0]+R[1]+1], \dots, M[2R[0]+R[1]]$ 
   */
18  $R[2] \leftarrow 0$ 
19 IF  $R[2] = R[0]$  GOTO 25
20  $R[3] \leftarrow R[0]+M[R[2]]$  /* pointer to tally of smaller items than current one */
21  $M[R[0]+R[1]+M[R[3]]] \leftarrow M[R[2]]$ 
22  $M[R[3]] \leftarrow M[R[3]]+1$ 
23  $R[2] \leftarrow R[2]+1$ 
24 GOTO 19

/* loop to copy sorted array to beginning of memory */
25  $R[2] \leftarrow 0$ 
26 IF  $R[2] = R[0]$  GOTO 29
27  $M[R[2]] \leftarrow M[R[0]+R[1]+R[2]+1]$ 
28  $R[2] \leftarrow R[2]+1$ 
29 HALT

```

Algorithm 2: MergeSort

```

/* locations  $M[R[0]], \dots, M[2R[0]-1]$  will be scratch space for merging */
/* stack will start at  $M[2R[0]]$  */
/* at each level of recursion, we'll store the interval  $[i, j)$  to be sorted
   and line number to goto when finished */
/*  $R[3]$  will be stack pointer,  $R[2]$  loop counter */
/* loop to allocate scratch space for merging */
1 IF  $R[2] = R[0]$ , GOTO 5
2 MALLOC
3  $R[2] \leftarrow R[2] + 1$ 
4 GOTO 1
   /* set up top of stack */
5 MALLOC; MALLOC; MALLOC
6  $R[3] \leftarrow R[0] + R[0]$  /* pointer to top of stack */
7  $M[R[3]] \leftarrow 0$  /* want to sort interval  $[0, R[0])$  */
8  $M[R[3] + 1] \leftarrow R[0]$ 
9  $M[R[3] + 2] \leftarrow 38$  /* ...then go to line 38 */
   /* Start of recursion */
10 IF  $M[R[3] + 1] = M[R[3]]$  OR  $M[R[3] + 1] = M[R[3]] + 1$ , GOTO  $M[R[3] + 2]$  /* base cases */
11 MALLOC; MALLOC; MALLOC /* extend stack to be safe */
   /* sort first half */
12  $M[R[3] + 3] \leftarrow M[R[3]]$ 
13  $M[R[3] + 4] \leftarrow M[R[3]] + \lfloor (M[R[3] + 1] - M[R[3]]) / 2 \rfloor$ 
14  $M[R[3] + 5] \leftarrow 17$ 
15  $R[3] \leftarrow R[3] + 3$ 
16 GOTO 10
   /* sort right half */
17  $M[R[3]] \leftarrow M[R[3] + 1]$ 
18  $M[R[3] + 1] \leftarrow M[R[3] - 2]$ 
19  $M[R[3] + 2] \leftarrow 21$ 
20 GOTO 10
   /* merge */
21  $R[3] \leftarrow R[3] - 3$ 
22  $R[2] \leftarrow 0$  /* number of elements merged */
23  $R[4] \leftarrow M[R[3]]$  /* location in left half */
24  $R[5] \leftarrow M[R[3] + 3]$  /* location in right half */
25 IF  $R[2] = M[R[3] + 1] - M[R[3]]$ , GOTO 34 /* test if done merging */
26 IF  $(M[R[4]] > M[R[5]]$  OR  $R[4] = M[R[3] + 3])$ , GOTO 30
   /* using element from left half */
27  $M[R[0] + R[2]] \leftarrow M[R[4]]$ 
28  $R[2] \leftarrow R[2] + 1$ ;  $R[5] \leftarrow R[5] + 1$ 
29 GOTO 25
   /* using element from right half */
30  $M[R[0] + R[2]] \leftarrow M[R[5]]$ 
31  $R[2] \leftarrow R[2] + 1$ ;  $R[4] \leftarrow R[4] + 1$ 
32 GOTO 25
   /* copy merged list back into input */
33  $R[2] \leftarrow 0$ 
34 IF  $R[2] = M[R[3] + 1] - M[R[3]]$ , GOTO  $M[R[3] + 2]$ 
35  $M[R[3] + R[2]] \leftarrow M[R[0] + R[2]]$ 
36  $R[2] \leftarrow R[2] + 1$ 

```

6.9 Computational Complexity

Fixing the model of computation, we can be precise about measuring efficiency:

Definition 6.7. The *running time of P on an input $x \in \mathbf{N}^*$* is the number t of steps before P reaches a halting configuration on x . That is, the smallest t for which there is a sequence $C_0 \Rightarrow_P C_1 \Rightarrow_P \cdots \Rightarrow_P C_t$ such that C_0 is the initial configuration of P on input x and C_t is a halting configuration (i.e. the program counter is $q + 1$, where q is the number of instructions in P).

Definition 6.8. The (worst-case) *running time of P* is the function $T : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ where $T(n, k)$ is the maximum running time of P over all inputs $x \in \{0, \dots, k\}^n$.

1. counting sort has running time $T(n, k) = O(n + k)$.
2. merge sort has running time $T(n, k) = O(n \log n)$.

A crude but important efficiency requirement is that the running time should grow at most polynomially in the input size:

Definition 6.9. P runs in *polynomial time* if there is a constant c such that P runs in time $T(n, k) = O((n \log k)^c)$.

- Note that word size can't get larger than $O(\log n + \log k)$.
- Most algorithm analyses allow word size $w = O(\log n)$ from start of computation (and assume $k \leq 2^w = \text{poly}(n)$).

Claim: this affects runtime by at most a constant factor. (why?)

Weaker and stronger requirements, respectively, are to allow a polynomial dependence on the magnitude of the numbers (rather than bit-length) or no dependence on the numbers at all:

Definition 6.10. P is *pseudopolynomial time* if there is a constant c such that P runs in time $T(n, k) = O((nk)^c)$.

Definition 6.11. P is *strongly polynomial time* if there is a constant c such that P runs in time $T(n, k) = O(n^c)$.

Q: What are examples of algorithms we've seen that are pseudopolynomial time but not polynomial time, and polynomial time but not strongly polynomial time?

Q: Why don't we allow memory cells to contain arbitrary (unbounded) integers, instead of bounded-length words?

How much does the input representation matter? Consider three choices for problem where the input consists of one or two n -bit numbers $N \in \mathbf{N}$ (e.g. addition, multiplication, or factoring):

- Put N in a single word of length $w = n$.
- Break N into n/w words of $w = \log n + O(1)$ bits each.
- Break N into n bits, put each bit in a separate word. Word size will still be $w = \log n + O(1)$ to allow indexing into input.

Q: How much can running time differ between these three cases? Will it matter for polynomial time? Strongly polynomial time?

References

[B47] George Boole. *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning*. Macmillan, Barclay, & Macmillan, Cambridge, England, 1847. Reprinted in Oxford by Basil Blackwell, 1951.

[S38] Claude E. Shannon. A symbolic analysis of relay and switching circuits. 57(12):713–723, December 1938.