6.1 A RAM Model

We need a precise model of computation to have a mathematically rigorous theory of computation, and especially to understand limits of computation. 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 $M = (M[0], \ldots, M[S-1])$ of w-bit *words*, which are also interpreted as numbers in the range $\{0, \ldots, 2^w - 1\}$
- We also have an array $R = (R[0], \ldots, R[r-1])$ 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: intepreting 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 $(M[0], \ldots, M[n-1])$.
  - 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]:

[1] For computer scientists, the natural numbers **N** include zero!

- $R[i] \leftarrow m$
- $R[i] \leftarrow R[j] + R[k]$
- $R[i] \leftarrow R[j] - R[k]$
- $R[i] \leftarrow R[j] \cdot R[k]$
- $R[i] \leftarrow R[j] \wedge R[k]$ (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 6.1. A *word-RAM program* is any *finite* sequence of instructions $P = (P_1, P_2, \ldots, P_q)$ 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.

6.2 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 6.2. A *configuration* of a word-RAM program P is a tuple $C = (\ell, S, w, R, M)$, where $\ell \in \{1, \ldots, q+1\}$ is the *program counter*, $S \in \mathbf{N}$ is the *space usage*, $w \in \mathbf{N}$ is the *word size*, $R = (R[0], \ldots, R[r-1]) \in \{0, \ldots, 2^w - 1\}^r$ is the *register array*, and $M = (M[0], \ldots, M[S-1]) \in \{0, \ldots, 2^w - 1\}^S$ is the *memory array*.

Definition 6.3. For a configuration $C = (\ell, S, w, R, M)$ of a word-RAM program $P = (P_1, P_2, \ldots, P_q)$, we define the *next configuration* $C' = (\ell', S', w', R', M')$, writing $C \Rightarrow_P C'$, as follows:

- if $P_\ell =$ "IF $R[i] = 0$, GOTO m" for some $i < r$ and $m \in \{1, \ldots, q\}$, 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 $i = 0, \ldots, S-1$.
- Otherwise (in particular if $P_\ell =$ 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 6.4. A *computational problem* is a function $f : \mathbf{N}^* \to 2^{\mathbf{N}^*}$.[2] $f(x)$ is the *set of correct answers* on input x.

Definition 6.5. A RAM program $P = (P_1 \ldots, P_q)$ with $r \geq 2$ registers *solves* the problem $f : \mathbf{N}^* \to 2^{\mathbf{N}^*}$ if for every input $x = (x_1, \ldots, x_n) \in \mathbf{N}^*$ and $k \geq \max\{x_1, \ldots, x_n, m\}$ where m is the largest constant occurring in P, there is a sequence of configurations $C_0, \ldots, C_t$ such that:

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

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

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

Some key points:

- Uniformity: we require that there is a single finite program should work for arbitrary inputs of unbounded size (as $n, k \to \infty$).
- 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.

6.3 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],...,M[R[0]-1].                                          */
   /* tallies will be kept in M[R[0]],...,M[R[0]+R[1]]                      */
   /* sorted array will be first be placed in M[R[0]+R[1]+1],...,M[2R[0]+R[1]],
      and copied at end to M[0],...,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] ← R[1]+1
 2 IF R[2] = 0, GOTO 6
 3 MALLOC
 4 R[2] ← 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]]] ← M[R[0]+M[R[2]]]+1
 8 R[2] ← 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] ← R[0]; R[3] ← 0
12 R[4] ← M[R[2]]
13 M[R[2]] ← R[3]
14 R[3] ← R[3]+R[4]
15 IF R[2] = R[1]+R[0] GOTO 18
16 R[2] ← R[2]+1
17 GOTO 12

   /* Loop to compute sorted array in locations M[R[0]+R[1]+1],...,M[2R[0]+R[1]]
      */
18 R[2] ← 0
19 IF R[2] = R[0] GOTO 25
20 R[3] ← R[0]+M[R[2]]   /* pointer to tally of smaller items than current one */
21 M[R[0]+R[1]+M[R[3]]] ← M[R[2]]
22 M[R[3]] ← M[R[3]]+1
23 R[2] ← R[2]+1
24 GOTO 19
   /* loop to copy sorted array to beginning of memory                     */
25 R[2] ← 0
26 IF R[2] = R[0] GOTO 29
27 M[R[2]] ← M[R[0]+R[1]+R[2]+1]
28 R[2] ← R[2]+1
29 HALT
```

Algorithm 2: MergeSort

```
   /* locations M[R[0]],...,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] ← R[2] + 1
 4 GOTO 1
   /* set up top of stack                                                    */
 5 MALLOC; MALLOC; MALLOC
 6 R[3] ← R[0] + R[0]                          /* pointer to top of stack */
 7 M[R[3]] ← 0                               /* want to sort interval [0,R[0]) */
 8 M[R[3]+1] ← R[0]
 9 M[R[3]+2] ← 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] ← M[R[3]]
13 M[R[3]+4] ← M[R[3]] + ⌊(M[R[3]+1] - M[R[3]])/2⌋
14 M[R[3]+5] ← 17
15 R[3] ← R[3] + 3
16 GOTO 10

   /* sort right half                                                        */
17 M[R[3]] ← M[R[3]+1]
18 M[R[3]+1] ← M[R[3]-2]
19 M[R[3]+2] ← 21
20 GOTO 10
   /* merge                                                                  */
21 R[3] ← R[3] - 3
22 R[2] ← 0                                 /* number of elements merged */
23 R[4] ← M[R[3]]                             /* location in left half */
24 R[5] ← 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]] ← M[R[5]]
28 R[2] ← R[2] + 1; R[5] ← R[5] + 1
29 GOTO 25
   /* using element from right half                                          */
30 M[R[0]+R[2]] ← M[R[4]]
31 R[2] ← R[2] + 1; R[4] ← R[4] + 1
32 GOTO 25

   /* copy merged list back into input                                       */
33 R[2] ← 0
34 IF R[2] = M[R[3]+1] - M[R[3]], GOTO M[R[3]+2]
35 M[R[3]+R[2]] ← M[R[0]+R[2]]
36 R[2] ← R[2] + 1
37 GOTO 34.
38 HALT
```

6.4 Computational Complexity

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

Definition 6.6. 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 6.7. The (worst-case) *running time of P* is the function $T : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ where $T(n,k)$ is the maximum running time of P over all inputs $x \in \{0,\ldots,k\}^n$.

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 6.8. 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 6.9. P is *pseudopolynomial time* if there is a constant c such that P runs in time $T(n,k) = O((nk)^c)$.

Definition 6.10. 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?