### 3.1 What is Computation?

How does one determine whether a function is “computable”? Before we answer this question, we define primitive recursive functions, which take natural numbers as inputs and return a natural number. There are three basic primitive recursive functions:

- **The constant function** $C()$.
- **The successor function** $\sigma(x) = x + 1$.
- **The projection function** $\pi(x_1, x_2, \ldots, x_n, i) = x_i$.

Additional primitive recursive functions are obtained by using the following two operations:

- **Composition**: Given a $k$-ary primitive function $f$, and $k$ $l$-ary functions $g_1, \ldots, g_k$, the composition of $f$ with $g_1, \ldots, g_k$ is the function $h(x_1, \ldots, x_l) = f(g_1(x_1, \ldots, x_l), \ldots, g_k(x_1, \ldots, x_l))$.

- **Recursion**: Given a $k$-ary primitive recursive function $f$ and a $(k + 2)$-ary function $g$, a $(k + 1)$-ary function is defined recursively as:

$$
\begin{align*}
  h(x_1, \ldots, x_k) &= f(x_1, \ldots, x_k) \\
  h(x_1, \ldots, x_k, y + 1) &= g(h(x_1, \ldots, x_k, y), x_1, \ldots, x_k, y)
\end{align*}
$$

If $f$ and $g$ are primitive recursive, then $h$ is also primitive recursive in both above cases.

Hence, function is **primitive recursive** if it is one of the basic functions above, or can be obtained from one of the basic functions by applying composition and recursion a finite number of times.

In order to define computable functions, we need to introduce the **unbounded minimization operator**. Let us see the relation of computability with loop controls.

Let $P(t, x_1, \ldots, x_n)$ be a primitive recursive predicate, that is, the characteristic function $\chi_p(t, x_1, \ldots, x_n)$, which is 1 (resp. 0) when $P(t, x_1, \ldots, x_n)$ is true (resp. false) is primitive recursive. Then

$$
  f(y, x_1, \ldots, x_n) = \min_{t \leq y} P(t, x_1, \ldots, x_n)
$$

is also primitive recursive. The operation “$\min_{t \leq y}$” is called **bounded minimalization**. This operation is similar to the traditional “for loop” in which one specifies in advance the number of times the loop will be executed.

However if we don’t specify the bound $y$ in the minimalization operator, we obtain unbounded minimalization

$$
  f(y, x_1, \ldots, x_n) = \min_t P(t, x_1, \ldots, x_n)
$$

Unbounded minimalization corresponds to “while loop” in standard programming programming languages. If there is no value of $t$ for which $P(t, x_1, \ldots, x_n)$ is true, then $f(x_1, \ldots, x_n)$ is undefined.
The set of computable functions is the set of functions that can be obtained from basic functions by applying composition, recursion and unbounded minimalization.

There are some computable functions which are not primitive recursive. Consider Ackerman’s function $A_k(x)$ defined as:

\[
\begin{align*}
A_1(x) &= 2x \\
A_k(1) &= 2^k \\
A_k(x) &= A_{k-1}(A_k(x-1))
\end{align*}
\]

It’s interesting to see how fast the function grows. Let us calculate the first few values, which will illustrate this:

\[
\begin{align*}
A_1(x) &= 2x \\
A_2(x) &= 2^x \\
A_3(x) &= 2^{2^{2^{\cdots}\times \text{x times}}}
\end{align*}
\]

### 3.1.1 Church’s Thesis

The model we considered is closed to standard programming languages. Even if we consider any other model of computation, like the Turing machine, we get the same rules for computability. Roughly speaking, Church’s Thesis states that the set of computable functions is independent of the model used. In fact, even in the newer models of computation — quantum computation and DNA computation, we obtain the same set of computable functions.

An example of an uncomputable function is the Halting Problem: Given a problem $p$ and an input $X$, determine whether $p$ halts on $X$. We can simulate $p$ on $X$, and if $p$ halts on $X$, we can return YES, but we do not know how long we should simulate $p$ before we conclude that $p$ does not halt.

If you want to know more about computability, CPS 140 (undergraduate) and CPS 240 (graduate) talk about it in more detail.

### 3.2 Algorithms

- We now introduce the basic notion of an algorithm.
  - **Algorithm**: A well-defined procedure that transfers an input to an output.
    * Not a program (but often specified like it): An algorithm can often be implemented in several ways.
  - **Design**: We will study methods/ideas/tricks for developing (fast!) algorithms.
  - **Analysis**: Abstract/mathematical comparison of algorithms (without actually implementing them).

An algorithm should give a consistent, correct answer each time it is applied to the same input. Also, it must halt no matter what input is given to it.

- Math is needed in three ways:
  - Formal specification of problem
  - Analysis of correctness
  - Analysis of efficiency (time, memory use,...)

- Hopefully the class will show that algorithms matter!
3.3 Performance of Algorithms

There are generally more than one possible ways to solve a problem. Then how do we decide which algorithm is “better”? There are two broad criteria for measuring the performance of an algorithm:

**Quality of output.** It depends on what the user wants. A good example is the popular internet site Map Quest, that tells you routes from sources to destinations. The path it gives may not be the shortest one, but easier to follow because the user wants a reliable, simple and fast route.

**Resources.** It is important to know how “expensive” the algorithm would be to implement and execute:

- **Time**: how much time does the algorithm take to get the output when given an input
- **Space**: how much memory it uses during execution
- **Processors**: in case of parallel processing, it matters how many processors you use
- **Communication**: in case of distributed systems or remote computing, one would want to minimize communication complexity (like time taken for message passing, memory access complications)
- **Energy**: one wouldn’t want to use up a lot of energy especially if you are working on lap tops or PDAs which need battery
- **Chip Area**: to make some applications faster, they are being hardwired, but that needs more VLSI chip area which is an expensive commodity for small machines.

We will mainly be looking at Time and Space, but may look at other criteria later.

3.4 Abstract Models of Computation

Before we go about calculating time and space required for an algorithm, we must define a model. There are two models we will be looking at:

**Random-access machine (RAM) model:**

1. Memory consists of infinite array
2. Instructions executed sequentially one at a time
3. All instructions take unit time:
   - Load/Store
   - Arithmetic (e.g., +, −, *, /)
   - Logic (e.g., >)

   - Running time of an algorithm is the number of RAM instructions it executes.

   - RAM model not completely realistic.
     - memory not infinite (even though we often imagine it is when we program)
     - not all memory accesses take same time (cache, main memory, disk, internet)
     - not all arithmetic operations take same time (e.g., multiplications are more expensive than additions)
     - instruction pipelining
     - other processes

   - But RAM model often enough to give relatively realistic results (if we don’t cheat too much).
Another model, which is more realistic and important for algorithms which are applied on huge amounts of data is the so called two-level I/O model.

**I/O model:**

- CPU exchanges information (data) at a fixed rate, say 10 nsec, with the main memory which has a fixed size of $M$
- The main memory exchanges information (data) in blocks of size $B$ with the disk which has infinite memory. Here the access time is much more, say 10 msec

![Figure 3.1: Two-level I/O model.](image)

In this model the space complexity of an algorithm is measured by the number of disk blocks used by the algorithm to store the data. The time complexity is measured by the number of block transfers (also called the number I/Os) performed by the algorithm.

### 3.5 Algorithm example: Insertion-sort

In class we looked at Selection-sort, but Insertion-sort shown here illustrates the same point.

#### 3.5.1 Specification

- Input: $n$ integers in array $A[1..n]$
- Output: $A$ sorted in increasing order
3.5.2 Insertion-sort algorithm

```
FOR j = 2 to n DO
  key = A[j]
  i = j - 1
  WHILE i > 0 and A[i] > key DO
    A[i + 1] = A[i]
    i = i - 1
  OD
  A[i + 1] = key
OD
```

Remark. The above box shows an example of the pseudo-code that we will sometimes use to describe algorithms.

```
5 2 4 6 1 3  j=2  i=1  key=2
5 5 4 6 1 3  i=0
2 5 4 6 1 3

2 5 4 6 1 3  j=3  i=2  key=4
2 5 5 6 1 3  i=1
2 4 5 6 1 3

2 4 5 6 1 3  j=4  i=3  key=6
2 4 5 6 1 3

2 4 5 6 1 3  j=5  i=4  key=1
2 4 5 6 6 3  i=3
2 4 5 5 6 3  i=2
2 4 4 5 6 3  i=1
2 2 4 5 6 3  i=0
4 2 4 5 6 3

1 2 4 5 6 3  j=6  i=5  key=3
1 2 4 5 6 6  i=4
1 2 4 5 5 6  i=3
1 2 4 4 5 6  i=2
1 2 3 4 5 6
```

Figure 3.2: Insertion sort.
3.5.3 Correctness

- Induction:
  - The Invariant “A[1..j − 1] is sorted” holds at the beginning of each iteration of FOR-loop.
  - When j = n + 1 we have correct output.

3.5.4 Analysis

- We want to predict the resource use of the algorithm.
- We can be interested in different resources
  - but normally running time.
- To analyze running time we need mathematical model of a computer:

- Running time of insertion-sort depends on many things
  - How sorted the input is
  - How big the input it
  - ...
- Normally we are interested in running time as a function of input size
  - in insertion-sort: n.
- We don’t really want to count every RAM instruction
  - Let us analyze insertion-sort by assuming that line i in the program use $c_i$ RAM instructions.
  - How many times are each line of the program executed?
    * Let $t_j$ be the number of times line 4 (the WHILE statement) is executed in the $j$th iteration.

<table>
<thead>
<tr>
<th>FOR $j = 2$ to $n$ DO</th>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>$key = A[j]$</td>
<td>$c_1$</td>
<td>n</td>
</tr>
<tr>
<td>$i = j - 1$</td>
<td>$c_2$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>WHILE $i &gt; 0$ and $A[i] &gt; key$ DO</td>
<td>$c_3$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>$A[i + 1] = A[i]$</td>
<td>$c_4$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>$i = i - 1$</td>
<td>$c_5$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>OD</td>
<td>$c_6$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>OD</td>
<td>$c_7$</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>

- Running time: (depends on $t_j$)

\[
T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_0 \sum_{j=2}^{n} (t_j - 1) + c_7 (n - 1)
\]
Best case: \( t_j = 1 \) (already sorted)

\[
T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)
\]

= \( k_1 n - k_2 \)

**Linear function of** \( n \)

Worst case: \( t_j = j \) (sorted in decreasing order)

\[
T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j - 1) + c_7 (n - 1)
\]

= \( c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_4 \left( \frac{n(n+1)}{2} - 1 \right) + c_5 \left( \frac{(n-1)n}{2} \right) + c_6 \left( \frac{(n-1)n}{2} \right) + c_7 (n - 1) \)

= \( (c_4/2 + c_5/2 + c_6/2)n^2 + (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7)n \)

= \( k_3 n^2 + k_4 n - k_5 \)

**Quadratic function of** \( n \)

“Average case”: Be careful! (average over what?)

We assume \( n \) numbers chosen randomly \( \Rightarrow t_j = (j+1)/2 \)

\[
T(n) = k_6 n^2 + k_7 n + k_8
\]

Still **Quadratic function of** \( n \)

Remarks.

- We will normally be interested in worst-case running time.
  - Upper bound on running time for any input.
  - For some algorithms, worst-case occur fairly often.
  - Average case often as bad as worst case (but not always!).

- We will only consider order of growth of running time:
  - We already ignored cost of each statement and used the constants \( c_i \).
  - We even ignored \( c_i \) and used \( k_i \).
  - We just said that best case was linear in \( n \) and worst/average case quadratic in \( n \).

\( \Rightarrow \) **O-notation** (best case \( O(n) \), worst/average case \( O(n^2) \))

### 3.6 Algorithms matter!

Let us look at another sorting algorithm, Mergesort. This also sorts \( n \) numbers in the same way as insertion sort, but it proceeds in the following manner:

- Divide \( n \) elements into two subsequences of \( n/2 \) elements each.
• Sort the two subsequences independently and recursively.
• Merge the two sorted subsequences.

We will look at this algorithm in more detail in the next lecture. The running time of this method can be calculated as follows:

\[ T(n) \leq c_1 + 2T(n/2) + c_2n \]

\[ \Rightarrow T(n) = 10n \log n \text{ (say)} \]

• Sort 1 billion \((10^9)\) integers on
  - 1 GHz computer (1000 million instructions per second) using \(n^2/2\) algorithm.
  - 100 MHz personal computer (100 million instructions per second) using \(50n \log n\) algorithm.

• Computer:
  \[
  \frac{(10^9)^2 \text{ inst.}}{2 \times 10^9 \text{ inst. per second}} = 5 \times 10^8 \text{ seconds} \approx 5000 \text{ days.}
  \]

Even if we use an ultrafast computer with \(10^{12}\) clockcycles per second, we will need 5 days.

• Personal computer:
  \[
  \frac{10 \cdot 10^9 \cdot \log 10^9 \text{ inst.}}{10^8 \text{ inst. per second}} < \frac{10 \cdot 10^9 \cdot 9 \cdot 3}{10^8} = 27 \times 10^2 \text{ seconds}
  \]
  which is about an hour.

### 3.7 I/O Complexity

Let us look at the significance of the I/O model that we introduced earlier. Suppose we want to multiply two large matrices \(P\) and \(Q\) each of size \(n \times n\). And let us assume that data is stored in row-major order, which means each row is stored sequentially.

Recall that matrix multiplication is defined as:

\[ r_{ij} = \sum_{k=1}^{n} p_{ik} \cdot q_{kj} \]

where \(R = P \times Q\).

Hence the \(i\)th row of \(P\) gets multiplied to the \(j\)th column of \(Q\). If the matrices are stored in row-major order, getting the row of \(P\) is no problem. One block of the \(i\)th row can be brought into the main memory in one I/O. But for matrix \(Q\), in the worst case, if the size of a block is less than \(n\), each inner multiplication would involve an I/O. So a naive program would take \(n^2\) I/Os. One way to get around this would be to find the transpose and store \(Q\) in column-major form (as \(Q^{-1}\) would be stored in its row major form). The best algorithm known till date to find the transpose takes

\[ \frac{n^2}{B \log_B n} \]

I/Os where \(B\) here is the block size that can be transferred in one I/O.