Chapter 1

Basics

1.1 Algorithms

Algorithms are systematic procedures for solving specific problems. Generically speaking, an algorithm can be described as taking an input, and after a sequence of operations produce an output. The relationship between input and output depends on the problem the algorithm solves.

Time complexity

In order to define time complexity of an algorithm one needs to define some parameters related to the problem an algorithm is trying to solve. For example, an algorithm for pricing an option might have the number of time periods as a parameter. Time complexity is given as a function of this parameter which measures the number of primitive operations (again depends on the problem) executed in getting from the input to the output.

Space complexity

Space complexity of an algorithm is the space required by an algorithm. For example, the space complexity of an algorithm for pricing a call option might be the space to hold the binomial model describing the movement of a stock.
proc $Min(array, n)$
begin
  $min = array[1]$
  for $i := 2$ to $n$ step 1 do
    if $min > array[i]$ then $min = array[i]$ fi
  od
  return($min$)
end

Figure 1.1: Find the minimum element in an array of $n$ elements

proc $Max(array, n)$
begin
  $max = array[1]$
  for $i := 2$ to $n$ step 1 do
    if $max < array[i]$ then $max = array[i]$ fi
  od
  return($max$)
end

Figure 1.2: Find the maximum element in an array of $n$ elements

Example

Suppose we are given an array of $n$ elements and we want to find the minimum and the maximum element in the array. One way is to call $MIN$ and $MAX$ (algorithms given in Figure 1.1 and 1.2). This algorithm has the time complexity $2n - 2$ (comparison is our primitive operation) and $n + 2$ space complexity. Consider the algorithm given in Figure 1.3.

Exercise 1.1.1 Prove that algorithm 1.3 finds the minimum and the maximum element. For ease of reasoning you can assume that the number of elements $n$ is odd. What is the time complexity of the algorithm? Use comparisons as your primitive operations.
\begin{verbatim}
proc MinMax(array, n)
    begin
        if odd(n)
            then max = array[1]
                 min = array[1]
                 start = 2
        else
            max = -\infty
            min = \infty
            start = 1
        fi
        for i := start to n - 1 step 2 do
            if array[i] < array[i + 1]
                then
                    if min > array[i] then min = array[i] fi
                    if max < array[i + 1] then max = array[i + 1] fi
                else
                    if min > array[i + 1] then min = array[i + 1] fi
                    if max < array[i] then max = array[i] fi
                fi
            od
        return(min, max)
    end
\end{verbatim}

Figure 1.3: Given an array array of n elements find the minimum and the maximum elements.
1.2 Notation

This section introduces some mathematical notation which will be used throughout the notes.

We will use iff as a short hand for if and only if. $[x]$ is the greatest integer $y$ such that $y \leq x$. $\lfloor x \rfloor$ is the least integer $y$ such that $y \geq x$. For example, $\lfloor 2.3 \rfloor = 2$ and $\lceil 3.14 \rceil = 4$. A vector will be written as $\vec{x}$ and the $i$-th element of vector $\vec{x}$ will be denoted by $\vec{x}[i]$. If it is clear from the context that we are dealing with a vector, we will not write an arrow over the symbol denoting the vector. A vector of size $n$ can also be regarded as a matrix of dimension $n \times 1$. Given two vectors $\vec{x}$ and $\vec{y}$, there dot product (denoted as $\vec{x} \cdot \vec{y}$) is defined as:

$$\sum_{i=1}^{n} \vec{x}[i] \vec{y}[i]$$

We will use $\mathbb{R}$ to denote the set of real numbers, $\mathbb{R}_{\geq 0}$ as the set of non-negative reals, and $\mathbb{R}_+$ as the set of positive reals. Given a real number $x$, $x^+$ is a shorthand for $\max\{0, x\}$.

1.3 Asymptotic Notation

Asymptotic notation is useful for studying relative efficiencies of algorithms in the limit. For example, suppose we are given an algorithm for pricing options. Asymptotic notation allows us to reason about such an algorithm as the number of time periods goes to infinity. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

$\Theta$-notation

For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions such that: $f(n) \in \Theta(g(n))$ iff there exists positive constants $c_1$, $c_2$, and $n_0$ such that for all $n \geq n_0$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Exercise 1.3.1 Prove that $f(n) \in \Theta(g(n))$ implies that $g(n) \in \Theta(f(n))$. 
1.3. ASYMPTOTIC NOTATION

Exercise 1.3.2 Suppose we are given a polynomial $g(n)$ of degree $k$, i.e., $g(n) = \sum_{i=0}^{k} c_i n^i$ where all the coefficients $c_i$ are non-negative. Prove that $g(n) \in \Theta(n^k)$.

Exercise 1.3.3 Prove that if $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists, then $f(n) \in \Theta(g(n))$.

O-notation

In order to talk about asymptotic upper bound we use the $O$ notation. Given a function $g(n)$, we say that $f(n) \in O(g(n))$ iff there exists positive constants $c$ and $n_0$ such that for all $n \geq n_0$

$$0 \leq f(n) \leq cg(n)$$

Exercise 1.3.4 Prove that if $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$, then $f(n) \in \Theta(g(n))$.

Ω-notation

Ω notation lets us talk about asymptotic lower bounds. Given a function $g(n)$, we say that $f(n) \in \Omega(g(n))$ iff there exists positive constants $c$ and $n_0$ such that for all $n \geq n_0$

$$0 \leq cg(n) \leq f(n)$$

Exercise 1.3.5 Prove that if $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(f(n))$, then $f(n) \in \Theta(g(n))$.

Exercise 1.3.6 Give a yes/no answer to the questions given below. Please justify your answer.
1. Is $2^n = O(2^{\sqrt{n}})$ ?
2. Is $2^n = O(2^\pi)$ ?
3. Is $n = O(\sqrt{n})$ ?

Exercise 1.3.7 Prove that $\log n = O(n^\epsilon)$ for any $\epsilon > 0$. 
1.4 Recurrences

A recurrence is an equation that describes a function in terms of its values on smaller inputs. Recurrences are very useful in defining the time or space complexity of various algorithms. We discuss two methods for solving recurrences in this section. Applications of recurrences for analyzing time and space complexity of algorithms will be provided later.

**Guessing method**

In this method we guess a solution and then use induction to prove that this indeed is the solution. Consider the following recurrence:

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(\lfloor \frac{n}{2} \rfloor) + n & \text{if } n > 1 
\end{cases} \]

We will prove that \( T(n) = O(n \log n) \) by induction. We will prove that for \( n \geq 2 \), \( T(n) \leq 2n \log n \) (check that this implies that \( T(n) = O(n \log n) \)).

- **Basis of induction**
  
  For \( n = 2 \), \( T(n) = 4 \) and the result is obviously true (check this).

- **Induction step**
  
  Consider \( n > 2 \) and assume that the result is true for all integers less than \( n \). Using the recurrence we get:

  \[
  T(n) \leq 2(2\lfloor \frac{n}{2} \rfloor \log(\lfloor \frac{n}{2} \rfloor)) + n 
  \leq 2n \log(\lfloor \frac{n}{2} \rfloor) + n \quad (2\lfloor \frac{n}{2} \rfloor \leq n) 
  \leq 2n \log n - 2n \log 2 + n \quad (\lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}) 
  \leq 2n \log n 
  \]
1.4. RECURRENCES

Iteration method

Consider the recurrence we solved before. First we will assume that \( n \) is a power of 2, i.e. \( n = 2^k \) and then give a proof for an arbitrary \( n \).

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T\left(\lfloor \frac{n}{2} \rfloor \right) + n & \text{if } n > 1
\end{cases}
\]

We have following series of equations:

\[
T(2^k) = 2T(2^{k-1}) + 2^k \\
= 2(2T(2^{k-2}) + 2^{k-1}) + 2^k \\
= 4T(2^{k-2}) + 2 \cdot 2^k \\
= 2^i T(2^{k-i}) + i \cdot 2^k \quad (\text{for } i \geq 1) \\
= 2^k T(1) + k \cdot 2^k \\
= (k + 1)2^k
\]

Now we consider the case of arbitrary \( n \). For an arbitrary \( n \) consider \( k = \lfloor \log n \rfloor \). It is easy to see that \( 2^{k-1} < n \leq 2^k \). Hence, \( T(n) \leq (k + 1)2^k \) (because the result is true for \( 2^k \)). Notice that this proves that \( T(n) \in O(n \log n) \) (check this argument).

Binary search

Algorithm given in Figure 1.4 searches for an element \( e \) in a sorted array of \( n \) integers. Consider the following invariant \( I \):

If element \( e \) occurs in \( \text{array} \) it is in the range from \( \text{start} \) to \( \text{end} \), i.e., \( e \) is in the set

\[
\{\text{array[start]}, \text{array[start + 1]}, \cdots, \text{array[end]}\}.
\]

We will prove that the invariant \( I \) holds each time the loop starts. It is obviously true at the beginning (why?). Suppose it is true when the while loop is entered (notice that \( \text{end} < \text{start} \) means that the set of possible elements that could be \( e \) is empty). If \( \text{array[middle]} = e \), we have found the element. If \( \text{array[middle]} < e \), then \( e \) could not be in the set

\[
\{\text{array[start]}, \cdots, \text{array[middle]}\}
\]

If \( \text{array[middle]} > e \), then \( e \) could not be in the set
proc BinarySearch(array, n, e)
    begin
        start := 1
        end := n
        middle := [(start + end)/2]
        while start ≤ end do
            (found the element)
            if array[middle] = e then return(middle) fi
            if array[middle] < e
                then start = middle + 1
            else end = middle - 1
            fi
        od
        (element e was not found)
        return(NULL)
    end

Figure 1.4: Binary search for finding an element in a sorted array of integers

(remember the array is sorted), so we set start = middle + 1. Notice that invariant I is still valid because e has to be in the set \{array[middle+1], \ldots, array[end]\} (if it occurs in the array at all). Reasoning for the case when array[middle] > e, is left as an exercise. This proves the correctness of the algorithm BinarySearch.

Exercise 1.4.1 Let T(n) be the time complexity of the algorithm given in Figure 1.4. Prove that T(n) satisfies the following recurrence:

\[
T(n) \leq \begin{cases} 
3 & \text{if } n = 1 \\
T(\lfloor n/2 \rfloor) + 2 & \text{if } n > 1 
\end{cases}
\]

The primitive operation here is comparison. Hint: Use proof by induction.

Exercise 1.4.2 Use induction to prove that T(n) ≤ clog n for some constant c and n ≥ 2. Find a suitable constant c.

Exercise 1.4.3 First, assume that n is a power of two, i.e., n = 2^k for some k. Find an expression for T(n) by the substitution method for the special case when n is a power of two. Argue that n = O(log n).
1.5. LINKED LIST

proc Search(L, k)
  begin
    x = head(L)
    while x ≠ NULL and x.key ≠ k do
      x = x.next
    od
    return(x)
  end

Figure 1.5: Return the node that contains k

1.5  Linked List

A linked list is a data structure in which the objects are arranged in a linear order. The order of elements in a link list is given by pointers associated with a node. Each node of a doubly linked list $L$ has three fields:

- *key* (value associated with that node)
- *next* (points to the next node in the order)
- *prev* (points to the previous node in the order)

For a node $n$, $n.next = NULL$ means that $n$ is the last node in the linked list. If $n.prev = NULL$, then $n$ is the first node in the linked list. The first element of the linked list is called the *head*. In a singly linked list component *prev* is missing.

Searching a Linked List

Given a value $k$, the code given in Figure 1.5 searches for a node $n$ such that $n.key = k$. The routine starts at the head of the linked list and marches down the linked list until it finds the specified element.

Inserting a node

The code for inserting a new node after a designated node in the linked list is given in Figure 1.6.

**Exercise 1.5.1** Write a psuedocode for a routine $InsertElement(L,k)$ which inserts a node with key value $k$ in the linked list $L$ if it is not al-
\textbf{proc} \textit{InsertNode}(x, after) \\
\textbf{begin} \\
\hspace{1em} temp = after.next \\
\hspace{1em} after.next = x \\
\hspace{1em} x.prev = after \\
\hspace{1em} x.next = temp \\
\textbf{end} \\

Figure 1.6: Insert node $x$ after node \textit{after}

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{insertion_example}
\caption{Insert a node into the linked list}
\end{figure}
ready present. **Hint**: use the routines *Search* and *InsertNode* presented earlier.

### 1.6 Stack

In a **stack**, the element deleted from the set is the one most recently inserted, or the stack implements a **last-in, first-out** (or **LIFO** policy). A stack supports following operations:

- **InitStack(size)**
  Initialize a stack with capacity *size* (maximum number of elements a stack can hold).

- **Push(S,k)**
  Push element *k* on top of the stack *S*.

- **Pop(S)**
  Pop the top element of the stack and return it.

**Exercise 1.6.1** Describe an implementation of stack using doubly linked lists.

### 1.7 Queue

In a **queue** the element deleted is always the one that has been in the set for the longest time: the queue implements a **first-in, first-out** or **FIFO** policy.

Following operations are supported by the queue:

- **InitQueue(size)**
  Initialize a queue with capacity *size* (maximum number of elements a queue can hold).

- **Enqueue(Q,k)**
  Queue an element *k* at the end of the queue *Q*.

- **Dequeue(Q)**
  Delete the element from *Q* that has been there for the longest time and return it.
Exercise 1.7.1 Describe an implementation of a queue using doubly linked lists.

1.8 Hash Table

If we use a linked list to represent a set, then finding out whether a particular element is in the set can in the worst case take time equal to the size of the set or linked list. **Hash Table** is a data structure that allows for fast searching. Let $K$ be the possible set of values or keys. A 1-dimensional **hash function** $h$ is function from $K$ to the set of integers from 1 to $n$ (set of integers from 1 to $n$ will be denoted by $[n]$), i.e. $h: K \rightarrow [n]$. A hash table $H$ supports the following operations:

- $Search(H,k)$
  Search whether the hash table $H$ has a node with the key $k$.

- $Add(H,k)$
  Add a node with key $k$ to the hash table $H$.

- $Delete(H,k)$
  Delete a node with key $k$ from the hash table $H$.

We discuss an implementation of hash tables using linked lists. A hash table $H$ can be implemented as an single dimensional array $HT$ of linked list, where $HT[i]$ is the linked list of nodes whose keys have hash value $i$. The search operation $Search(H,k)$ involves the following two steps:

- First, we compute the hash value $h(k)$ of the $k$.

- Second, we search the linked list $HT[h(k)]$ to find out whether a node with key $k$ is in the linked list.

Implementation of $Add(H,k)$ and $Delete(H,k)$ operations is left as an exercise.

Exercise 1.8.1 Explain in detail how the operations $Add(H,k)$ and $Delete(H,k)$ are implemented for a hash table $H$.  

1.9. DIRECTED GRAPHS

Notice that the hash function $h$ maps a key to a single value, i.e., the function $h$ is single dimensional. For many applications, it is more natural to have a multi-dimensional hash table, i.e., $h(k)$ is a $r$-dimensional vector $(i_1, \cdots, i_r)$. In this case a hash table can be implemented as an array of linked lists of dimension $r$. In this case, the search operation $Search(H,k)$ is implemented as follows:

- First, we compute the hash value $h(k) = (i_1, \cdots, i_r)$ of the $k$. Note that in this case the hash value is a vector of size $r$.

- Second, we search the linked list $HT[i_1][\cdots][i_r]$ to find out whether a node with key $k$ is in the hash table.

Implementation of $Add(H,k)$ and $Delete(H,k)$ operations is left as an exercise.

1.9 Directed Graphs

A directed graph $G$ has two components $(V, E)$ where

- $V$ is the set of vertices. If $n = |V|$, then without loss of generality we will assume that $V = \{1, 2, \cdots, n\}$.

- $E$ is the set of edges. Formally, $E \subseteq V \times V$ and $(u, v) \in E$ means that there is an edge from $u$ to $v$ in the graph.

Consider a directed graph $G = (V, E)$. Successors and predecessors of node $v$ are denoted by $succ(v)$ and $pred(v)$ respectively. Formally these sets are defined as

$$
succ(v) = \{u \mid (v, u) \in E\}
$$

$$
pred(v) = \{u \mid (u, v) \in E\}
$$

Next we discuss two representations of directed graphs.
**Adjacency Matrix**

Consider a graph $G = (V, E)$ with $n$ vertices. An adjacency matrix for graph $G$ (denoted as $A(G)$) is a $n \times n$ matrix described as follows:

$$A(G)[i, j] = \begin{cases} 
1 & \text{if } (i, j) \in E \\
0 & \text{otherwise}
\end{cases}$$

The adjacency matrix $A(G)$ for graph $G$ shown in Figure 1.8 is

$$\begin{pmatrix}
0 & 1 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{pmatrix}$$

**Adjacency list**

Given a graph $G = (V, E)$ with $n$ vertices we can represent it as a vector of linked lists $\vec{L}$ (size of the vector is $n$) such that $\vec{L}[i]$ is the linked list of successors of node $i$. The adjacency list for the graph $G$ (depicted in Figure 1.8) is shown in Figure 1.9.

**Undirected Graph**

A directed graph $G = (V, E)$ is called **undirected** iff $E \subseteq V \times V$ is **symmetric**, i.e., $(i, j) \in E$ implies that $(j, i) \in E$. One can view
1.9. DIRECTED GRAPHS

Successors of node 1

Successors of node 2

Successors of node 3

Figure 1.9: Adjacency list representation of graph in Figure 1.8

Figure 1.10: An undirected graph.
undirected graph as a graph where the edges have no direction. An example, of an undirected graph $G$ is shown in Figure 1.10.

Adjacency matrix $A(G)$ of the graph $G$ shown in Figure 1.10 is:

$$
\begin{pmatrix}
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{pmatrix}
$$

Notice that adjacency matrix $A(G)$ of an undirected graph $G$ is always symmetric, i.e., $A(G)[i, j] = 1$ implies that $A(G)[j, i] = 1$.

**Directed Acyclic Graphs**

Given a graph $G = (V, E)$, a sequence of nodes $u_0, u_1, \ldots, u_k$ is a path through $G$ if and only if $(u_i, u_{i+1}) \in E$ for all $0 \leq i \leq k - 1$. A path $u_0, u_1, \ldots, u_k$ is called a cycle if and only if $u_0 = u_k$. A directed graph $G = (V, E)$ is called a directed acyclic graph (or DAG for short) if and only if $G$ has no cycles.

**Exercise 1.9.1** Given a graph $G = (V, E)$ a node $v \in V$ is called a sink if and only if it has no successors. Prove that if $G$ is a DAG it must have at least one sink. Hint: Assume that a graph $G$ does not have a sink and then show that $G$ has a cycle.

**Exercise 1.9.2** Based on exercise 1.9.1 provide an algorithm to detect whether a graph $G = (V, E)$ is a DAG. Hint: Find all sinks in $G$ (if there are none $G$ is not a DAG). Remove these sinks from $G$ and repeat the process.

A DAG $G = (V, E)$ is called a tree if it satisfies following conditions:

- There is a distinguished node root such that $\text{pred}(\text{root})$ is empty.
- All other nodes have precisely one predecessor. Formally, $v \neq \text{root}$ implies that $|\text{pred}(v)| = 1$.
- A node with no successors is called a leaf.

**Exercise 1.9.3** Prove that a tree $G = (V, E)$ has $|V| - 1$ edges and there is an unique path from the root to any node.
1.9. DIRECTED GRAPHS

We introduced trees as special case of DAGs. Next we give an implementation of trees. Each node in a tree has the following fields:

- *key* (information associated with the node)
- *children* (linked-list of children of this node)
- *parent* (unique parent of this node)
- *next, prev* (used to implement a linked-list)

A tree has a distinguished node *root* which has no parent (*parent* field of *root* is *NULL*). Notice that a leaf does not have any children so the *children* field will be *NULL*. Traversing trees in a certain systematic way is a key concept. There is a large class of solutions that are based on traversing trees in a systematic fashion. We introduce a procedure to traverse a tree in *post order* and print out the nodes as they are visited. Traversing the tree in post order prints the sub-trees of a node (trees rooted at the children of the node) before printing out the node itself. A recursive procedure for printing out a tree in post-order is shown Figure 1.11. *PostOrder(root)* prints a tree in post order.

**Exercise 1.9.4** Pre-order traversal of a tree visits a node before visiting the sub-trees rooted at its children. Write the psuedo-code to print out a tree in pre-order.

**Exercise 1.9.5** Given a tree with *n* nodes what is the time complexity of *PostOrder*?

**Weighted Directed Graphs**

A *weighted directed graph* $WG$ (or *WDG* for short) has three components $WG = (V, E, W)$, where $V$ and $E$ denote the set of vertices and edges respectively and $W : E \rightarrow \mathbb{R}$ assigns real weights to edges. $W(u, v)$ is weight of the edge $(u, v)$. A *WDG* $WG = (V, E, W)$ is called a *probabilistic directed graph* (or *PDG* for short) if it satisfies the following conditions:

- $W(u, v) \geq 0$ for all $(u, v) \in E$.  

**Figure 1.11:** Print a tree in post order

- For all $u \in V$ such that $\text{succ}(u) \neq \emptyset$ ($u$ has successors) we have

$$
\sum_{v \in \text{succ}(u)} W(u, v) = 1
$$

In other words, the sum of the weight of all edges emanating out of a node is 1. In this case $W(u, v)$ can be interpreted as the probability of going from edge $u$ to $v$.

Since edge weights in a probabilistic directed graph represent the transition probabilities, we write the edge weights $W(u, v)$ as $p(u, v)$ to illustrate the fact that edge weights are actually probabilities. Therefore, when we write a weighted directed graph as $(V, E, p)$, $p$ should be interpreted as transition probabilities rather than edge weights. A PDG $WG = (V, E, p)$ is called a probabilistic directed acyclic graph or a lattice) if $(V, E)$ is a DAG. Figure 1.12 shows a lattice.

**Markov Chain**

A Markov chain with $n$ states $\{1, 2, \ldots, n\}$ has a $n \times n$ transition matrix $Q$ associated with it. The transition matrix has the following properties:

- $Q[i, j] \geq 0$ for all $i$ and $j$, i.e., the entries of $Q$ are non-negative.
1.9. DIRECTED GRAPHS

- $\sum_{j=1}^{n} Q[i, j] = 1$, i.e., each row of $Q$ sums up to one. If $\sum_{j=1}^{n} Q[i, j] < 1$, then the chain is called **sub-Markov**. $Q[i, j]$ is the probability of transitioning from state $i$ to state $j$.

Notice that a Markov chain with $n$ states and transition matrix $Q$ can be represented as a probabilistic directed graph $PDG = (V, E, p)$ such that $V = \{1, 2, \cdots, n\}$, $E = V \times V$, and $p(i, j) = Q[i, j]$. 

Figure 1.12: A lattice.