Course Notes for *Algorithms*

Fei Li*

*Email: fei.li.best@gmail.com.

Contents

1 Running Time

Tight bound $\Theta(g(n))$ For a given function $g(n)$, we denote by $\Theta(g(n))$ the set

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constant } c_1, c_2 \text{ and } n_0 \text{ such that}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\},$$

and if $f(n) \in \Theta(g(n))$ we just write $f(n) = \Theta(g(n))$. We say $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Upper bound $O(g(n))$ We use $O(g(n))$ to denote the set

$$O(g(n)) = \{f(n) : \text{there exist positive constant } c \text{ and } n_0 \text{ such that}$$
$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\},$$

and we write $f(n) = O(g(n))$ if $f(n) \in O(g(n))$. We say $g(n)$ is an ***asymptotically upper bound*** for $f(n)$.

Lower bound $\Omega(g(n))$ We use $\Omega(g(n))$ to denote the set

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constant } c \text{ and } n_0 \text{ such that}$$
$$f(n) \ge cg(n) \ge 0 \text{ for all } n \ge n_0\},$$

and we write $f(n) = \Omega(g(n))$ if $f(n) \in \Omega(g(n))$. We say $g(n)$ is an ***asymptotically lower bound*** for $f(n)$.

1.1 The Substitution Method

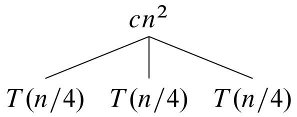
In substitution method, we make a guess of the running time, and substitute the guess into the recursion formula, to show that it works.

1.2 Recursion Trees

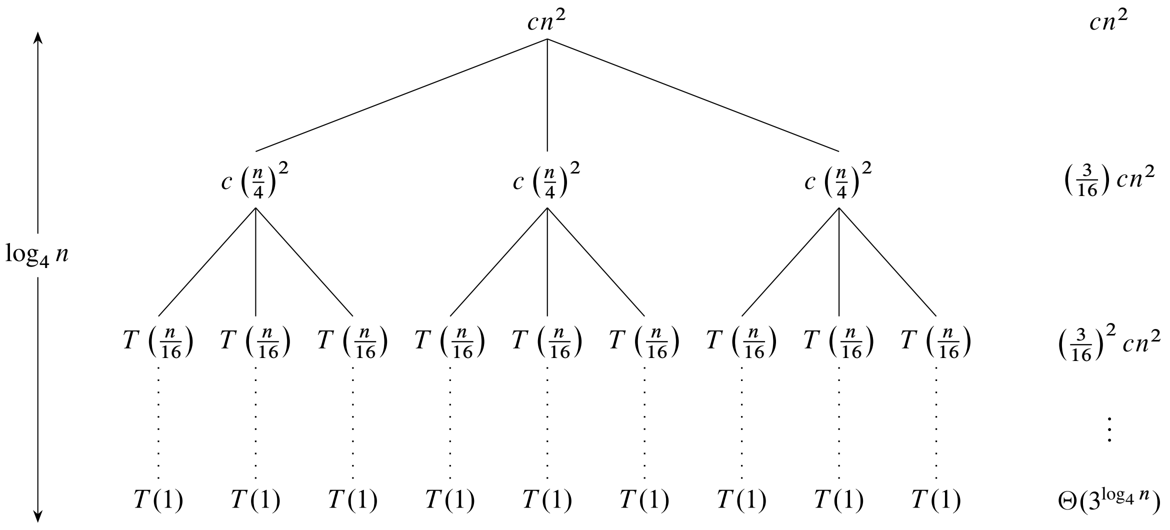
We illustrate the recursion tree method by examining the recursion

$$T(n) = 3T(n/4) + cn^2.$$

The running time when the input size is n is cn^2 plus 3 times the running time of input size $n/4$. The recursion tree is



We can expand $T(n/4)$ using the recursion formula, to obtain


cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$\left(\frac{3}{16}\right)cn^2$
$\log_4 n$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$
$T(1)$
$T(1)$
$T(1)$
$T(1)$
$T(1)$
$T(1)$
$T(1)$
$T(1)$
$\Theta(3^{\log_4 n})$

Since the size of the problem is reduced by a factor of 4 in each recursive call, the height of the tree is $\log_4 n$, and so the number of leaves is $3^{\log_4 n}$. At the first level there is cn^2 amount of work. At the second level there is $3 \times c\left(\frac{n}{4}\right)^2$ amount of work, at the third level there is $3 \times 3 \times c\left(\frac{n}{16}\right)^2$ amount of work...We sum them up to obtain the total time, using $3^{\log_4 n} = n \log_4 3$ (to verify apply $\log_4(\cdot)$ to both sides)

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&\le \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)}cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13}cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2).
\end{aligned}$$

Since we have to do at least cn^2 amount of work when the input size is n, we have $T(n) = \Theta(n^2)$.

In summary, in recursion tree method we expand the recursion formula into a tree, then sum over times at each level to obtain the total running time.

1.3 The Master Method

Theorem 1.1 (Master Theorem)**.** *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined by the recurrence*

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then

(1) If $f(n) = O(n^{(\log_b a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(2) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

(3) If the following two conditions hold:

(a) $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ for some constant $\epsilon > 0$;

(b) $af(n/b) \le cf(n)$ *for some constant* $c < 1$ *and all sufficiently large* n,

then $T(n) = \Theta(f(n))$.

Before going into the proof, let's see some examples about how we can apply the master theorem.

$T(n) = 9T(n/3) + n$ We have $a = 9, b = 3$, and $f(n) = n$, so $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{2-1})$, we can apply the first case of Theorem 1.1 to conclude that the solution is $T(n) = \Theta(n^2)$. The $a = 9$ is so large that the amount of work *increases* at each level $(n, \left(\frac{9}{3}\right)n, \left(\frac{9}{3}\right)^2 n, \ldots)$. If a were 3, so that $T(n) = 3T(n/3) + n$, then the second case applies: the amount of work at each level is n, and the height of the recursion tree is $\lg n$, so the total running time is $\Theta(n \lg n)$. If a were smaller than 3, for example $a = 2$, then $\log_3 2 < 1$ so the third case applies, and we would have $T(n) = \Theta(n)$. When the number of repetitive works ($a = 2$) is smaller than the decrease in problem size ($b = 3$), the amount of work at each level would *decrease* $(n, \left(\frac{2}{3}\right)n, \left(\frac{2}{3}\right)^2 n, \ldots)$ exponentially, so when we sum them up we can use the familiar geometric series argument to bound the running time by $\Theta(n)$.

$T(n) = T(2n/3) + 1$ We have $a = 1, b = 3/2, f(n) = 1$, and $n^{\log_b a} = n^0 = 1$. The second case applies and we conclude that the solution is $T(n) = \Theta(\lg n)$.

$T(n) = 3T(n/4) + n \lg n$ This is an example where case 3 applies. We have $a = 3, b = 4, f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. We know $f(n)$ is asymptotically lower bounded by n, so we can take $\epsilon \approx 0.2$. The regularity condition $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n$ also holds, so by the third case the solution is $T(n) = \Theta(n \lg n)$.

$T(n) = 2T(n/2) + n \lg n$ This is an example where the master method does not apply. $n^{\log_b a} = n$, but $f(n) = n \lg n$ is not polynomially larger than n^ϵ for any positive constant ϵ. On the other hand, it is easy to guess that the running time is $\Theta(n \lg^2 n)$.

$T(n) = 2T(n/2) + \Theta(n)$ Since $n^{\log_a} = n$, case 2 applies and the running time is $\Theta(n \lg n)$.

$T(n) = 8T(n/2) + \Theta(n^2)$ **and** $T(n) = 7T(n/2) + \Theta(n^2)$ The first equation and the second equation have slightly different a's. For the first equation, $n^{\log_b a} = 3$ so the first term dominates and we have $T(n) = \Theta(n^3)$ by case 1. For the second equation, $n^{\log_b a} = \lg 7 \approx 2.8 > 2$, so again the first term dominates and we have $T(n) = \Theta(n^{\lg 7})$. We mention that this is the running time of the Strassen's algorithm for matrix multiplication.

Proof of the Master Theorem. For simplicity we only prove for the case when $T(n)$ is an exact power of b. We use the idea of recursion tree to solve for $T(n)$: at each level j, the problem size is reduced to n/b^j, and the work is $f(n/b^j)$. Each node has a children, the number of nodes at each level is a^j, so the total amount of work at level j is $a^j f(n/b^j)$. The height of the tree is $\log_b n$, so the total work is

$$T(n) = \Theta(a^{\log_b n}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j). \tag{1}$$

Let $g(n)$ denote the second term in Eq. (1).

Case 1 When the condition $f(n) = O(n^{(\log_b a)-\epsilon})$ in (1) holds, we substitute n/b^j for n to obtain $f(n/b^j) = O((n/b^j)^{(\log_b a)-\epsilon})$, so that

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{(\log_b a)-\epsilon}\right).$$

The sum in the parenthesis is

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{(\log_b a)-\epsilon} &= n^{(\log_b a)-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{(\log_b a)-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j \\
&= n^{(\log_b a)-\epsilon} \left(\frac{b^{\epsilon \log_b n}-1}{b^\epsilon-1}\right) \\
&= n^{(\log_b a)-\epsilon} \left(\frac{n^\epsilon-1}{b^\epsilon-1}\right) \\
&= n^{(\log_b a)-\epsilon} O(n^\epsilon) \\
&= O(n^{\log_b a})
\end{aligned}$$

so $g(n) = O(n^{\log_b a})$. We see from Eq. (1) that $T(n) = \Theta(n^{\log_b a})$.

Case 2 When the condition $f(n) = \Theta(n^{\log_b a})$ in (2) holds, we have $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$, so that

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right).$$

The sum in the parenthesis is

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\
&= n^{\log_b a} \log_b n \\
&= \Theta(n^{\log_b a} \lg n)
\end{aligned}$$

so $g(n) = \Theta(n^{\log_b a} \lg n)$. From Eq. (1) we have $T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3 From the conditions in (3),

$$\begin{gathered}
af(n/b) \le cf(n) \\
\Downarrow \\
f(n/b) \le (c/a)f(n) \\
\Downarrow \\
f(n/b^j) \le (c/a)^j f(n) \\
\Downarrow \\
a^j f(n/b^j) \le c^j f(n)
\end{gathered}$$

for n large enough. So

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\ &\le \sum_{j=0}^{\log_b n-1} c^j f(n) + O(1) \\ &\le f(n) \sum_{j=0}^{\infty} c^j + O(1) \\ &= f(n) \left(\frac{1}{1-c} \right) + O(1) \\ &= O(f(n)). \end{aligned}$$

We can conclude from Eq. (1) that in this case $T(n) = \Theta(f(n))$. □

2 Data Structures

2.1 Elementary Data Structures

2.1.1 Stacks and Queues

Stacks We can implement a stack of at most n elements with an array $S[1\,..\,n]$. The array has attribute $S.\mathit{top}$ that indexes the most recently inserted element. The stack consists of elements $S[1\,..\,S.\mathit{top}]$, where $S[1]$ is the element at the bottom of the stack and $S[S.\mathit{top}]$ is the element at the top. When $S.\mathit{top} = 0$, the stack is empty. If we attempt to pop an empty stack, we say the stack *underflows*, while if $S.\mathit{top}$ exceeds n, the stack *overflows*. We can implement the stack operations as in Algorithm 2.1.1.

Algorithm 2.1.1 Stacks

```
STACK-EMPTY($S$)
    if $S.top == 0$ then
        return TRUE
    else
        return FALSE

PUSH($S, x$)
    $S.top = S.top + 1$
    $S[S.top] = x$

POP($S$)
    if STACK-EMPTY($S$) then
        error "underflow"
    else$S.top = S.top - 1$
        return $S[S.top + 1]$
```

Queues We can implement a queue of at most $n - 1$ elements using an array $Q[1\,..\,n]$. The queue has attribute $Q.\mathit{head}$ that points to the first element in the queue, and $Q.\mathit{tail}$ that points to the last empty slot where a newly arriving element will be inserted.

We can imagine that when we dequeue, we move $Q.\mathit{head}$ to the right, and when we enqueue we move $Q.\mathit{tail}$ to the right. If any of them is already at the end of the array, then we move the pointer to the first slot of the array. See Algorithm 2.1.2 for implementation.

2.1.2 Linked Lists

Linked list is a data structure where we can conveniently insert and delete elements. The list L has one attribute $L.\mathit{head}$ that points to the head of the list. If the list is empty then $L.\mathit{head} = \text{NIL}$. It has the following methods: LIST-SEARCH, LIST-INSERT, and LIST-DELETE. Each element $x \in L$ is an object with three attributes:

$$x.\mathit{key},\, x.\mathit{prev} \text{ and } x.\mathit{next}\,.$$

If $x.\mathit{prev} = \text{NIL}$ then x is the head, while if $x.\mathit{next} = \text{NIL}$ it is the tail. See Algorithm 2.1.3 for implementation of linked list. Here is the running time for linked list's methods:

- The LIST-SEARCH procedure takes $\Theta(n)$ time. To search an element, we start from the head and go through the chain of the links.

Algorithm 2.1.2 Queue

```
ENQUEUE($Q, x$)            ✍ Add $x$ to the end of the queue.
    $Q[Q.tail] = x$
    if $Q.tail \neq len(Q)$ then
        $Q.tail = Q.tail + 1$
    else
        $Q.tail = 1$

DEQUEUE($Q$)            ✍ Remove the first element of the queue.
    $x = Q.head$
    if $Q.head \neq len(Q)$ then
        $Q.head = Q.head + 1$
    else
        $Q.head = 1$
    return $x$
```

- The LIST-INSERT procedure takes $O(1)$ time. To insert an element, we put it at the head.
- The LIST-DELETE procedure takes $O(1)$ time. If we want to delete a given key, then we have to first search in the list and then do delete. This would take $\Theta(n)$ time. To delete an element, we link $x.\mathit{prev}$ and $x.\mathit{next}$ together.

Linked list can be very useful. We shall see in Section 2.4 that linked list can be used in hash tables to solve collision. We can also use the idea of linked list to represent trees. For binary tree, we can implement a class T, with attribute $T.\mathit{root}$ that points to the root of the tree, similar to $L.\mathit{head}$, and with methods like SEARCH, INSERT and DELETE (we discuss these methods in detail in the context of binary search trees). Each node x inserted in the tree is an object with four attributes: $x.\mathit{key}$ for storing data, $x.p$ that points to the parent,$x.\mathit{left}$ that points to its left child and $x.\mathit{right}$ that points to its right child. To implement a tree in which a node can have an arbitrary number of children, we can substitute $x.\mathit{left}$ and $x.\mathit{right}$ with $x.\mathit{left\text{-}child}$ and $x.\mathit{right\text{-}sibling}$. We mention that other implementations are possible, for example heaps. Which scheme is the best depends on the application.

2.2 Binary Search Trees

Binary search tree is another data structure in which we can store data and perform *search*, *insert* and *delete* operations. In order to achieve these operations in an effective way, we attempt to link the data in some way, specifically into binary search trees. We put the data in nodes and we link the nodes so that they are organized into a binary tree T. Each node x is an object that has four attributes, see Table 1. We require the binary-search-tree property

$$x.\mathit{left}.\mathit{key} \leq x.\mathit{key} \leq x.\mathit{right}.\mathit{key}$$

to hold for any $x \in T$ after any modification of the tree. This property is how we organize the data in the tree, and it is this property that can let us perform efficient operations like search or insert.

We will see that the running time for the search, insert and delete operations are all proportional to the height of the tree. If the binary tree is balanced, so that $h = \lg n$, then those operations will be very efficient. However, if the tree is very unbalanced so that $h = \Theta(n)$, then those operations we wish to perform would take significantly longer time. There is no guarantee that when we insert elements a binary search tree will remain balanced. In Section 2.3 we will see how we can maintain the balance using red-black trees.

Algorithm 2.1.3 Linked List

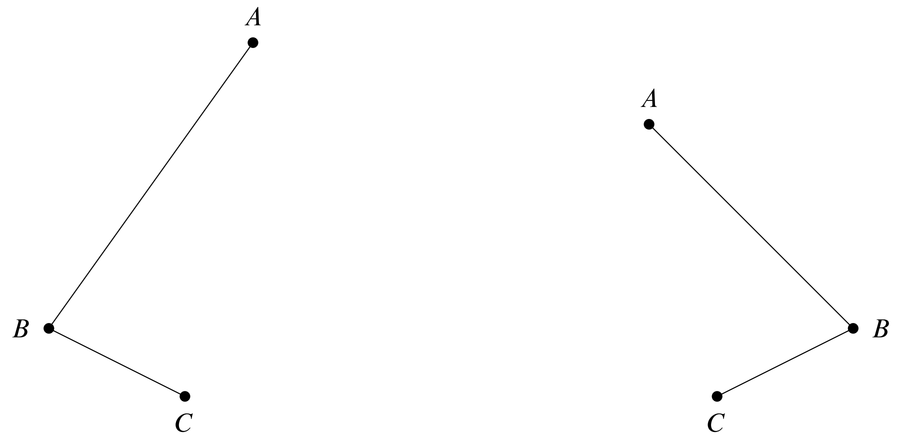
```
LIST-SEARCH($L, k$)
    $x = L.head$
    while $x \neq$ NIL and $x.key \neq k$ do
        $x = x.next$
    return $x$

LIST-INSERT($L, x$)
    $x.next = L.head$
    if $L.head \neq$ NIL then          ☙ If $L$ is not empty
        $L.head.prev = x$
    $L.head = x$
    $x.prev =$ NIL

LIST-DELETE($L, x$)
    if $x.prev ==$ NIL then          ☙ If $x$ is head
        $L.head = x.next$
    else          ☙ If $x$ is not head
        $x.prev.next = x.next$
    if $x.next \neq$ NIL then          ☙ If $x$ is not tail
        $x.next.prev = x.prev$
```

$x.key$	data
$x.p$	parent of x
$x.left$	left child of x
$x.right$	right child of x

Table 1: Attributes of a node in binary search tree.



(a) $B < C < A$. B is the minimum.

(b) $B > C > A$. B is the maximum.

Figure 1: Maximum and minimum points in branches in BST.

Note that in a binary search tree, for a “left triangle” pattern like Fig. 1a, the “peak” is the node with minimal key among the three nodes, while for a “right triangle” like Fig. 1b, the “peak” is the maximum among the three nodes. This observation will be helpful when we delete a node x and need to organize other nodes around x so as to maintain the binary-search-tree property.

We can print out the data in order in a BST using INORDER-TREE-WALK that takes $\Theta(n)$ time.

```
INORDER-TREE-WALK($x$)
    if $x \neq$ NIL then
        INORDER-TREE-WALK($x.left$)
        print($x.key$)
        INORDER-TREE-WALK($x.right$)
```

2.2.1 Queries

Search To search whether a key value k is in the tree T, we call TREE-SEARCH($T.root, k$). We start from the root and compare k with $T.root$. If k is smaller we go left and compare it to the left subtree, because all elements in the right subtree is larger than the root, so there is no chance that k will be in the right subtree. Similarly if $T.root < k$ then we go right and compare it to the right subtree.

```
TREE-SEARCH($x, k$)
    if $x ==$ NIL or $k == x.key$ then
        return $x$
    if $k < x.key$ then
        return TREE-SEARCH($x.left, k$)
    else
        return TREE-SEARCH($x.right, k$)
```

The nodes encountered during the recursion form a simple path downward from the root of the tree, thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

Maximum and Minimum To find the minimum element in T, we just need to keep going left. To find the maximum, we keep going right. Both of these procedures take $O(h)$ time since again we trace a simple path downward from the root.

```
TREE-MINIMUM($x$)
    while $x.left \neq$ NIL do
        $x = x.left$
    return $x$
```

```
TREE-MAXIMUM($x$)
    while $x.right \neq$ NIL do
        $x = x.right$
    return $x$
```

Successor and predecessor The successor of a node x is the node whose key is the smallest among all nodes that have keys larger than $x.key$. In other words,

$$succ(x) = \min\{y \mid x.key < y.key\}.$$

Similarly, the predecessor is

$$pred(x) = \max\{y \mid y.key < x.key\}.$$

If the node x has right subtree, then the successor of x would be the minimum element in the right subtree. If, however, x does not have a right subtree, then the successor is at the top of x and so we should walk along the up and right direction of the tree. As long as x is the right child of its parent $x.p$, the parent is smaller than x. We continue walking along the tree until the child is a *left* child of its parent (in other words we are able to *turn right*). By that time the parent should be larger than all nodes in its left subtree, and in particular, x.

```
TREE-SUCCESSOR($x$)
    if $x.right \neq$ NIL then
        return TREE-MINIMUM($x.right$)
    $y = x.p$
    while $y \neq$ NIL and $x == y.right$ do
        $x = y,\ y = y.p$          ▷ go upward
    return $y$
```

The algorithm for finding predecessor is symmetric. If the left subtree of x is not empty, then we return the maximum element of its subtree. If it does not have a left subtree, then we walk upward. As soon as we are able to *turn left*, we find the element that is smaller than all elements in its right subtree, and in particular, x. This element would then be the predecessor of x.

```
TREE-PREDECESSOR($x$)
    if $x.left \neq$ NIL then
        return TREE-MAXIMUM($x.left$)
    $y = x.p$
    while $y \neq$ NIL and $x == y.left$ do
        $x = y,\ y = y.p$          ▷ go upward
    return $y$
```

2.2.2 Insertion and Deletion

Insertion To insert an element z into T, we start from the root and walk down the tree to find the appropriate place, so that the binary-search-tree property is maintained. If $T = \varnothing$, then we let z be the root. Otherwise, if T is not empty, then we compare $z.key$ with $T.root.key$:

1. If $z.key < T.root.key$, then we should insert z somewhere in the left subtree of $T.root$.
2. If $z.key \geq T.root.key$, then we should insert somewhere in the right subtree of $T.root$.

In the first case we continue to compare z with the root's left child, and in the second case we compare z with the root's right child. We continue to do so, until we have no node to compare. In other words, we might arrive at a node x where we want to compare z with $x.left$, but find $x.left = \text{NIL}$. In this case we can let z be the left child of x. Or we might want to compare z with $x.right$ but find $x.right = \text{NIL}$. In this case we let z be the right child of x.

```
TREE-INSERT($T, z$)
    $x = T.root$, $y = \text{NIL}$
    while $x \neq \text{NIL}$ do
        $y = x$            ☙ After the iteration $y$ is the parent of $x$
        if $z.key < x.key$ then
            $x = x.left$            ☙ Go down the tree
        else
            $x = x.right$            ☙ Go down the tree
    $z.p = y$            ☙ $x$ becomes NIL. We find the node to append $z$
    if $y == \text{NIL}$ then
        $T.root = z$
    else if $z.key < y.key$ then
        $y.left = z$            ☙ Set $z$ as the left child of $y$
    else
        $y.right = z$            ☙ Set $z$ as the right child of $y$
```

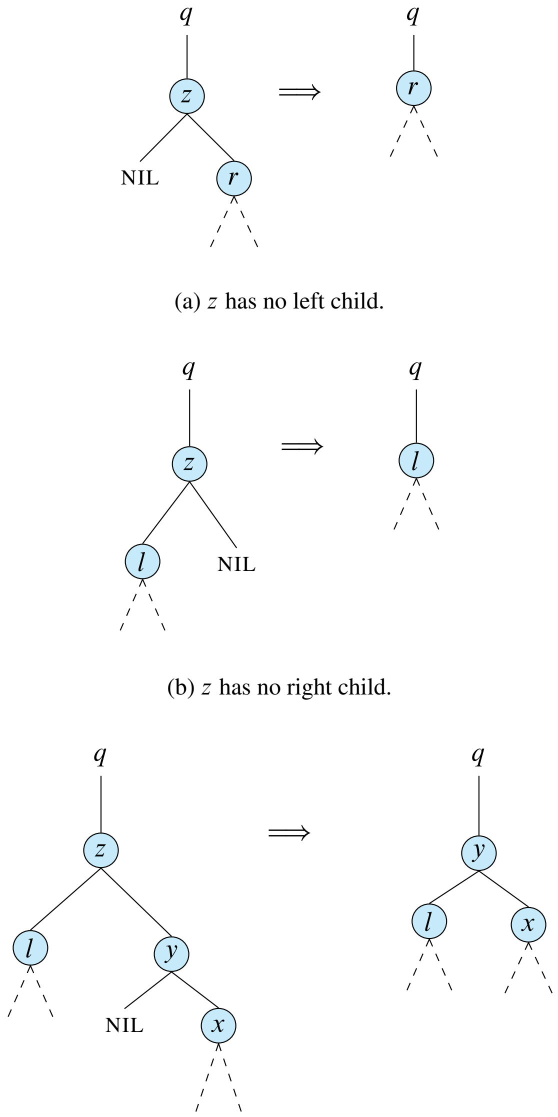
Notice that there is a small redundancy in the procedure TREE-INSERT, namely by the time x becomes NIL, we should know whether it is the left child or right child (of y) that is empty, but since we do not keep track of this information, after we find an empty spot we have to re-compare z with its to-be parent y.

Since we're tracing down the tree, the algorithm runs in $O(h)$ time.

Deletion To delete the node $z \in T$, we change the "connections" of its "surroundings". We need to consider four cases.

1. If z has no left child, then we just replace z with its right child (Fig. 2a).
2. Similarly, If z has no right child, then we just replace z with its left child (Fig. 2b).
3. If z has both left child and right child, then we replace z by its successor, y. Note that by its nature y should have no left child.
   (a) If its successor $y \in T$ is z's right child, then we replace z by y and then give z's left child to y (Fig. 2c).
   (b) If its successor $y \in T$ is not z's right child, but somewhere in the right subtree, we again replace z by y, but we should not forget to also replace y by its right child (Fig. 3).

In Cormen et al. 2009, the authors used a routine called TRANSPLANT. This routine replaces a node u by v, but notice that it does not handle u's children.



(a) z has no left child.

(b) z has no right child.

(c) The successor of z is its right child y.

Figure 2: Deletion in binary search tree.

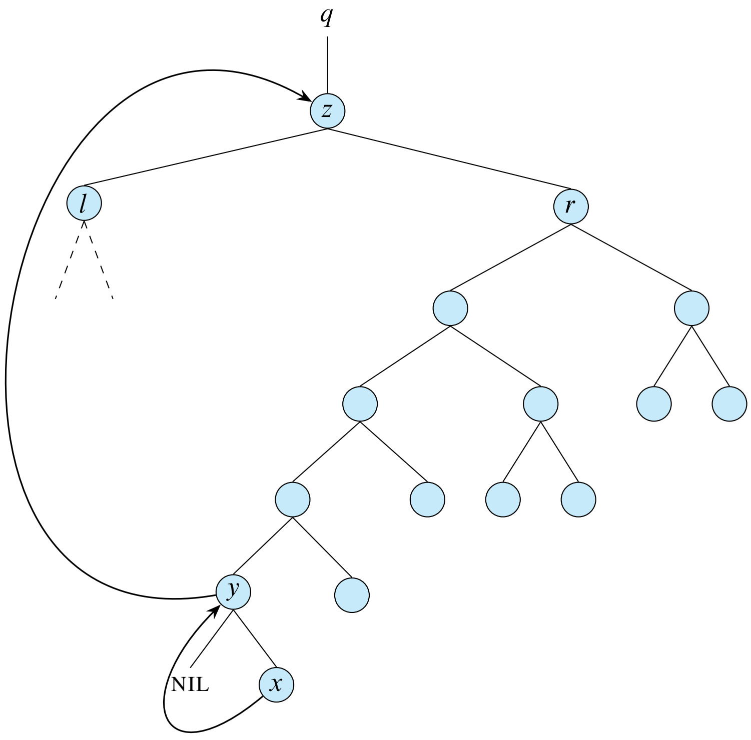


Figure 3: Delete the node $z \in T$ when its successor $y \in T$ lies within the right sub-tree of z. In this case we replace z by y and we replace y by its right child x (which may be NIL).

```
TRANSPLANT($T, u, v$)
  if $u.p$ == NIL then                    ❧ if $u$ is the root, then just set the root to $v$.
    $T.root = v$
  else if $u$ == $(u.p).left$ then        ❧ i.e. if $u$ is the left child of its parent,
    $(u.p).left = v$                      ❧ then replace $u$ by $v$.
  else                                    ❧ i.e. if $u$ is the right child of its parent,
    $(u.p).right = v$                     ❧ then replace $u$ by $v$.
  if $v \neq$ NIL then
    $v.p = u.p$                           ❧ set $v$'s parent as $u$'s parent.
```

In the delete procedure, if z's successor y is not its right child, then we need to transfer z's right subtree to y, while if the successor y is the right child of z, then we do not need to do this, since the right subtree of z is already the tree rooted at y. But in both cases we need to transfer z's left subtree to y. Since we used TREE-MINIMUM in deletion, the running time is again $O(h)$.

```
TREE-DELETE($T, z$)
  if $z.left$ == NIL then                 ❧ (1) if $z$ has no left child,
    TRANSPLANT($T, z, z.right$)           ❧ transplant $z$'s right subtree to its position.
  else if $z.right$ == NIL then           ❧ (2) if $z$ has no right child,
    TRANSPLANT($T, z, z.left$)            ❧ transplant $z$'s left subtree to its position.
  else
    $y$ = TREE-MINIMUM($z.right$)         ❧ find the successor of $z$
    if $y.p \neq z$ then                  ❧ (4) if $y$ is not the right child of $z$,
      TRANSPLANT($T, y, y.right$)         ❧ replace $y$ by its right child,
      $y.right = z.right$                 ❧ handle the right subtree of $z$,
      $(y.right).p = y$                   ❧ handle the right subtree of $z$.
    TRANSPLANT($T, z, y$)                 ❧ (3), (4) handle the parent of $z$
    $y.left = z.left$                     ❧ (3), (4) handle the left subtree of $z$
    $y.left.p = y$                        ❧ (3), (4) handle the left subtree of $z$
```

2.3 Red-Black Trees

From Section 2.2, we know that all operations in binary search tree, like searching, finding maximum or minimum element, insertion and deletion, have running time $O(h)$ that is proportional to the height of the tree. If the tree is balanced, then the height is $O(\lg n)$ so we can do all the operations in logarithmic time. But in a plain BST there is no guarantee that during insertions the tree will stay balanced. A red-black tree is a data structure that ensures the balance of the tree, and at the same time achieves $O(\lg n)$ time for all operations. The trade-off is that insertion and deletion are far more complicated than vanilla binary search tree.

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. It is required to maintain the following properties during the operations:

1. Every node is either red ♠ or black ♠.
2. The root is black ♠. Every leaf (NIL) is black ♠.
3. If a node is red ♠, then both its children are black ♠.
4. For each node, all simple paths from the node to descendant leaves contain the same number of black ♠ nodes.

Item 3 and Item 4 together constrain the tree to be balanced. If there is no Item 3 but only Item 4, then all nodes could be red and the red-black tree would turn into a plain BST. On the other hand, all nodes could be black in a red-black tree, but if we only insert black nodes, then Item 4 could be violated, so we may have to insert red nodes in many circumstances.

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the *black-height* of the node, denoted by $bh(x)$. We define the black height of a red-black tree to be the black height of its root.

Theorem 2.1. *A red-black tree with n internal nodes (namely, nodes that are not leaves) has height at most* $2\lg(n+1)$.

Proof. We first show that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes. We prove by induction on the height of x. If the height of x is 0, x is a leaf (NIL), and indeed the subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes. Suppose x has positive height. Each child of x has a black height of either $bh(x)$ or $bh(x) - 1$, depending on whether x is red or black. By hypothesis each child of x has at least $2^{bh(x)-1} - 1$ internal nodes. Thus the subtree rooted at x contains at least

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$$

internal nodes.

Let h be the height of the tree. From the properties of red-black tree (Item 3), at least half the nodes on any simple path from the root to a leaf, not counting the root, must be black. Consequently the black-height of the root must be at least $h/2$. Thus

$$n \geq 2^{h/2} - 1 \quad \Rightarrow \quad h \leq 2\lg(n+1).$$

□

2.4 Hash Tables

Hash table is convenient when we want to store some keys of size K out of a big universe U of keys. Instead of creating a huge space of size U, which is often impractical, we can use only $\Theta(K)$ space to store keys that will arrive in, while at the same time still maintain $O(1)$ time for the operations of search, delete and insert.

We maintain a hash table $T[0\,..\,m-1]$ with size $m << |U|$, and we use a hash function $h : U \to \{0, 1, \ldots, m-1\}$ to map the universe of keys to their slots in $T[0\,..\,m-1]$. We assume the computation of $h(k)$ takes $O(1)$ time for any $k \in U$. We say k *hashes* to $h(k)$ and $h(k)$ is the *hash value* of k. To put an element x into the hash table, we first compute the hash value of its key, $h = h(x.key)$, and then we put x to $T[h]$. Of course, with m much smaller than $|U|$, we have the possibility of collision. A first way to solve collision is *chaining*. We let each slot in $T[0\,..\,m-1]$ to be a linked list, and we put all elements that hash to the same slot into the same linked list, so as to distinguish between them. With chaining, search, insertion and deletion operations are easy to implement.

```
CHAINED-HASH-SEARCH($T, k$)
    LIST-SEARCH ($T[h(k)]$, $k$)

CHAINED-HASH-INSERT($T, x$)
    $h = h(x.key)$
    LIST-INSERT ($T[h]$, $x$)

CHAINED-HASH-DELETE($T, x$)
    $h = h(x.key)$
    LIST-DELETE ($T[h]$, $x$)
```

Let's analyze the running time of the operations. Insertion and deletion take $O(1)$ time, since inserting and deleting an element in a (doubly) linked list take $O(1)$ time (see Section 2.1.2). Now what is the time for searching in a hash table with chaining? It is obvious that the search time for k depends on the length of the list $T[k]$. Assume there are n elements and m slots. We define the *load factor* as $\alpha = n/m$. We assume *simple uniform hashing*, meaning that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. The expected number of elements for each linked list is then $n/m = \alpha$. For an unsuccessful search, it is clear that the average time is $\Theta(1+\alpha)$. (1 for computing the hash value). For an successful search, it can take slightly less time, but the average time is also $\Theta(1+\alpha)$. Below is a precise analysis.

Denote the n elements in the hash table by $\{x_1, \dots, x_n\}$, and denote the search time for x_i by S_i. We assume each element is equally likely to be searched for, so we shall calculate the average of expected search time for each element. What is S_i? It is the number of iterations in search for k_i, i.e. S_i is the number of elements in the list $T[h(k_i)]$ that lie before x_i, i.e. the elements that are later inserted into the hash table who all have the same hash value as k_i. Let $X_{ij} := I\{h(k_i) = h(k_j)\}$. We have $\mathbb{P}[h(k_i) = h(k_j)] = 1/m$ so $\mathbb{E}[X_{ij}] = 1/m$. We then have $S_i = X_{i(i+1)} + \cdots + X_{in}$. The average expected running time is then

$$
\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}(1+\mathbb{E}[S_i]) &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = \frac{1}{n}\left(n+\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= 1+\frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m} \\
&= 1+\frac{1}{n}\left[\frac{1}{m}\cdot(n-1)+\frac{1}{m}\cdot(n-2)+\cdots+\frac{1}{m}\cdot(n-n)\right] \\
&= 1+\frac{1}{nm}\left[(n-1)+\cdots+(n-n)\right] \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{nm}\left(\sum_{i=1}^{n} n-\sum_{i=1}^{n} i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) = 1+\frac{n-1}{nm} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n} = \Theta(1+\alpha).
\end{aligned}
$$

If m is proportional to n, namely for larger n hash table with larger size m is used, we have $n = O(m)$ and so $\alpha = n/m = O(m)/m = O(1)$. To summarize, *under the conditions*

1. *chaining is used to resolve collision;*
2. *each key is equally likely to hash into any slots, independently of other keys;*
3. *load factor is constant,*

searching in the hash table takes constant time.

2.4.1 Hash functions

Here we discuss several basic hash functions.

Uniformly distributed keys If the keys are uniformly distributed in $[0, 1]$, then the hash function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of simple uniform hashing.

Division method The division method takes an integer and return its *reminder* when divided by m:

$$h(k) = k \bmod m.$$

For example, if $m = 73$, then $h(34) = 34$ and $h(4211) = 50$. Note that if we represent the integer k in binary number as

$$b_q \cdots b_1 = b_q \cdot 2^{q-1} + \cdots + b_1 \cdot 2^0$$

where $b_j \in \{0, 1\}$, $j = 1, \ldots, q$, we see that dividing k by 2^p for some $p < q$ will leave with the remainder $b_{p-1} \cdots b_1 = b_{p-1} \cdot 2^{p-1} + \cdots + b_1 \cdot 2^0$, the lowest p bits of k. Thus we should avoid choosing m as a power of 2.

Multiplication method The multiplication method would first multiply k by some constant $A \in (0, 1)$, then extract the fractional part (($kA \bmod 1$) would extract the fractional part), which is again a number in $(0, 1)$, and then multiply m and take the floor, so as to return a number in $\{0, \ldots, m-1\}$. The formula is

$$h(k) = \lfloor m(kA \bmod 1) \rfloor, \qquad 0 < A < 1.$$

Setting $m = 2^p$ for some integer p makes the computation faster. Suppose the word size of the machine is w bits and $k \in (0, 2^w)$ is a w-bit integer. For example

- if $w = 2$ then $2^w = 2^2 = 100$ so $0 < k < 100$ is a 2-bits integer;
- if $w = 3$ then $2^w = 2^3 = 1000$ so $0 < k < 1000$ is a 3-bits integer;
- if $w = 4$ then $2^w = 2^4 = 10000$ so $0 < k < 10000$ is a 4-bits integer.

Additionally assume $A = s/2^w$, where $s \in (0, 2^w)$. Then $ks \in (0, 2^{2w})$ is a $2w$-bits integer. Denote the left w digits by $r_1 \in (0, 2^w)$ and the right w digits by $r_0 \in (0, 2^w)$, so that

$$ks = r_1 \cdot 2^w + r_0.$$

Then

$$kA = \frac{ks}{2^w} = \frac{r_1 \cdot 2^w + r_0}{2^w} = r_1 + \frac{r_0}{2^w}$$

so the fractional part of kA is $\frac{r_0}{2^w}$. We then multiply $m = 2^p$ to get

$$\left(\frac{r_0}{2^w}\right) \cdot 2^p.$$

What is this? Since r_0 is w-bits, we imagine that dividing it by $2^w = 1\,\underbrace{00\cdots0}_{w \text{ digits}}$ moves the decimal point to the left, all the way to the front of the number, and then multiplying by 2^p moves the decimal points to the right by p digits. Taking the floor, we discard what is at the right of the decimal point and only retain the left part, the first p digits of r_0.

So to summarize, when we use $m = 2^p$, to compute $h(k)$ we just need to select a w-bits integer s, multiply k by s and take the first p digits of the right w-bits of the product.

2.4.2 Open Addressing

We saw that in chaining we can solve collision by putting a linked list container in each hash table slot. *Open addressing* is a different strategy for solving collision. If a key is hashed to an occupied slot, then we try to find another empty place in the hash table to place the key. This is called *probing*. The hash function now becomes $h : U \times \{0, 1, \cdots, m-1\} \to \{0, 1, \cdots, m-1\}$. Each time we insert or search for a key we produce a sequence

$$(h(k, 0), h(k, 1), \ldots, h(k, m-1)),$$

which is required to be a permutation of $(0, 1, \ldots, m-1)$, so that every hash-table position is eventually considered. We introduce three methods for probing:

- *Linear Probing*:

  $$h(k, i) = (h'(k) + i) \bmod m,$$

  where $h' : U \to \{0, 1, \ldots, m-1\}$ is an ordinary hash function. So we first probe $T[h'(k)]$, then we probe $T[h'(k)+1], T[h'(k)+2]$ and so on...and we wrap around to slots $T[0], T[1]$... until $T[h'(k)-1]$. This is just the ordinary hash, plus the simple procedure that if a slot is occupied then check the next slot.

- *Quadratic Probing*:

  $$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m.$$

  In quadratic probing, the interval between probes increases quadratically.

- *Double Hashing*:

  $$h(k, i) = (h_1(k) + i h_2(k)) \bmod m.$$

  The starting position and the step are both determined by hash functions, so it is more “random".

We have different methods for probing but the procedures for insert and search are the same. For insertion, we examine elements j in the sequence $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ one by one until we find an empty slot $T[j]$, and then we insert the key there.

```
HASH-INSERT($T, k$)
  $i = 0$
  while $i \le m - 1$ do
    $j = h(k, i)$
    if $T[j]$ == NIL then
      $T[j] = k$
      return $j$
    else
      $i = i + 1$
  error “hash table overflow”
```

For searching, we again examine elements j in the sequence $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ one by one. We can stop our search if we find that $T[j]$ is empty, because k would have been inserted there.

```
HASH-SEARCH($T, k$)
  $i = 0$
```

```
while $i \le m - 1$ do
    $j = h(k, i)$
    if $T[j] == k$ then
        return $j$
    else if $T[j] ==$ NIL then
        return NIL
    $i = i + 1$
return NIL
```

It should be clear that probing makes deletion difficult. Suppose our hash table slots consist of $0, 1, 2, 3$ and 4, and we inserted k to slot 4 using the sequence $(0, 1, 2, 3, 4)$. So $0, 1, 2, 3$ are all occupied at the time. Then if we delete the key in, say, 2, then when we search for k in the hash table using $(0, 1, 2, 3, 4)$, we would find $T[2] ==$ NIL and the above function would return NIL.

The solution would be to mark the deleted slot as DELETED instead of None. We insert k to places that are empty or are marked as DELETED. The search function does not need to be modified. However, search times no longer depend on the load factor α, and indeed we suspect that it may be linear. For this reason chaining is more commonly used as a collision resolution technique when keys must be deleted.

Analysis of Open Addressing Looking at the two programs for insert and search, it seems at first that the running times are both linear in m, i.e. $O(m)$. However, it may take less than m iterations for searching or insertions, and that is the point of our analysis. We highlight two important assumptions that we are using in the following analysis:

- α is constant;
- we assume uniform hashing.

So for example if there are n elements in a hash table of size m, then when we search for a key k, with probability approximately $\alpha = n/m$ our first probe is unsuccessful; with probability approximately α^2 our second probe is unsuccessful.....if k is not in the table then the search time is bounded by $1 + \alpha + \alpha^2 + \cdots = 1/(1-\alpha)$. If k is in the table then the search time depends on when k was inserted. We worked out below that the average time is bounded by $(1/\alpha)\ln(1/(1-\alpha))$.

The assumption of uniform hashing is that the probe sequence $(h(k,0), h(k,1), \ldots, h(k, m-1))$ is equally likely to be any permutation of $(0, 1, \ldots, m-1)$. None of the above three probing methods is uniform. Linear probing and quadratic probing are only able to produce $\Theta(m)$ sequences, and for double hashing it is $\Theta(m^2)$, since each possible pair $(h_1(k), h_2(k))$ yields a distinct probe sequence.

Theorem 2.2. *Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an* unsuccessful *search is at most $1/(1-\alpha)$, assuming uniform hashing.*

Proof. In an unsuccessful search, we probe X occupied slots before finding an empty slot, from which we conclude that the key we are searching for is not in the table. We'd like to calculate $\mathbb{E}[X]$. There is a formula for this:

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=0}^{\infty} i \cdot \mathbb{P}\{X = i\} \\ &= \sum_{i=0}^{\infty} \big(\mathbb{P}\{X \ge i\} - \mathbb{P}\{X \ge i+1\}\big) \\ &= \sum_{i=1}^{\infty} \mathbb{P}\{X \ge i\}.\end{aligned}$$

So we need to calculate $\mathbb{P}\{X \geq i\}$. Let A_i be the event that the ith probe finds an occupied slot. Then $\{X \geq i\} = A_1 \cap A_2 \cap \cdots \cap A_i$. The probability is

$$\mathbb{P}\{A_1\} \cdot \mathbb{P}\{A_2|A_1\} \cdot \mathbb{P}\{A_3|A_1 \cap A_2\} \cdots \mathbb{P}\{A_i|A_1 \cap \cdots \cap A_{i-1}\}.$$

There are n elements and m slots so $\mathbb{P}\{A_1\} = n/m$. For $j > 1$, the probability is $(n - j + 1)/(m - j + 1)$, since we are finding (assume uniform hashing) among the remaining $(n - (j - 1))$ elements in one of the $(m - (j - 1))$ slots, given that the first $j - 1$ slots are found to be occupied. So we have

$$\begin{aligned}\mathbb{P}\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \\ &\leq \left(\frac{n}{m}\right)^i \\ &= \alpha^i.\end{aligned}$$

Consequently

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{\infty} \mathbb{P}\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^i \\ &= \frac{\alpha}{1-\alpha}.\end{aligned}$$

The number of probes in an unsuccessful search is $X + 1$, so the expectation is bounded by $1 + \alpha/(1 - \alpha) = 1/(1 - \alpha)$. □

If the hash table is half full, the average number of probes in an unsuccessful search is at most $1/(1 - 0.5) = 2$. If it is 90 percent full, the average number of probes is at most $1(1 - 0.9) = 10$. If α is constant, then $1/(1 - \alpha)$ is constant, so an unsuccessful search runs in $O(1)$ time.

We can see from the above HASH-INSERT function that inserting an element into an open-address hash table is basically the same as an unsuccessful search. So if α is constant, then insertion also takes $O(1)$ time.

Theorem 2.3. *The expected number of probes in a* successful *search is at most*

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}.$$

(assuming uniform hashing and assuming that each key in the table is equally likely to be searched for)

Proof. Denote S to be the number of probes in a successful search. The sequence of probes for k is the same as when k is inserted, i.e. $(h(k,i))_{i=0,1,\ldots,m-1}$. What is the upper bound for expected search time S_k for a particular key k? This depends on the load factor α when k was inserted. If k is the $(i + 1)$st key inserted into the hash table, so that the load factor was i/m, then an upper bound for $\mathbb{E}[S_k]$ is $1/(1 - i/m) = m/(m - i)$. By law of iterated expectation,

$$\begin{aligned}\mathbb{E}[S] = \mathbb{E}[\mathbb{E}[S_k]] \leq \frac{1}{n}\sum_{i=1}^{n-1} \frac{m}{m-i} &= \frac{m}{n}\sum_{i=1}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} \frac{1}{k} \leq \frac{1}{\alpha}\int_{m-n}^{m} \frac{1}{x} dx \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}.\end{aligned}$$

□

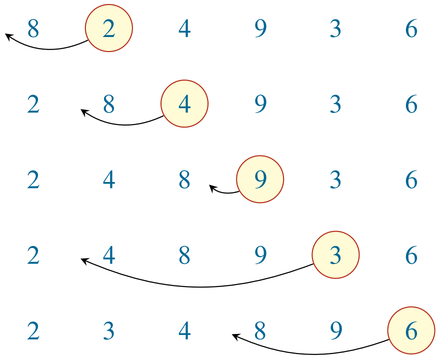


Figure 4: Illustration of insertion sort.

3 Sorting

3.1 InsertionSort

Insertion sort is a relatively straightforward sorting algorithm. Given an array A, it maintains a sorted subarray on the left part of A, expands it by comparing and swapping the next element with each element inside the sorted subarray. See Fig. 4.

```
INSERTION-SORT($A$)
    for $j = 2$ to $A.length$ do
        $key = A[j]$
        $i = j - 1$
        while $i > 0$ and $key < A[i]$ do        ✍ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$
            $A[i+1] = A[i]$
            $i = i - 1$
        $A[i+1] = key$
```

It is clear from Fig. 4 that the running time of insertion sort is $\Theta(n^2)$. Even if we use binary search when we insert the key, we still have to do the swaps, so in terms of swaps the complexity would still be $\Theta(n^2)$.

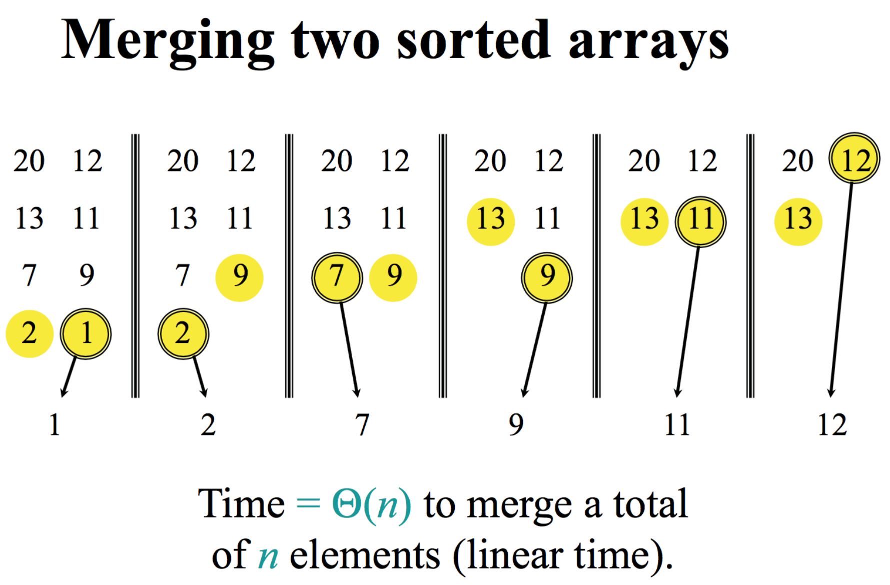
3.2 MergeSort

Merge sort is a divide and conquer algorithm. To sort an array of length n, the algorithm divides the array into two subarrays of length $n/2$, and recursively sort the two subarrays. It then combines the two sorted subarrays using a *merge* operation. This merge operation is the key. It is like a "finger-pointing" operation, illustrated in Fig. 5. To merge two subarrays, we imagine aligning them vertically and using two fingers to point at the two heads. For the one that is smaller, that element comes down and we move that finger upward. Continue to compare the elements at our two fingers this way until all elements come down. It is clear from this scheme that the running time of merge is $O(n)$.

Let $T(n)$ be the running time of merge sort for an array of size n. We have the relationship

$$T(n) = T(n/2) + T(n/2) + cn,$$

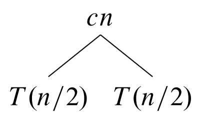
namely the time to sort A is the sum of the time spent on sorting left and right parts of A plus the time for merging the two parts, which takes $\Theta(n)$ times. We can use a recursion tree to get a picture of this equation:



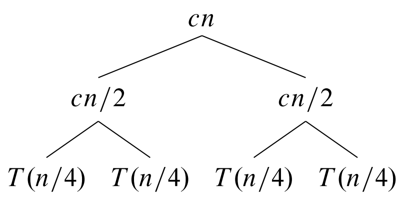
Merging two sorted arrays

Time = $\Theta(n)$ to merge a total of n elements (linear time).

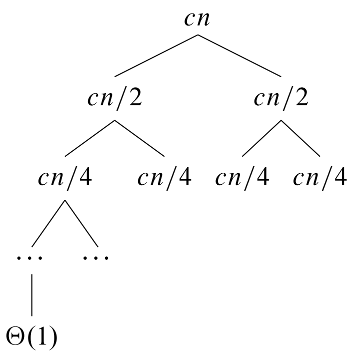
Figure 5: Illustration of merge sort. Image from MIT6.006 *Introduction to Algorithms* course slides.



The picture means that $T(n)$ is cn plus $T(n/2)$ and $T(n/2)$. We can then further decompose each $T(n/2)$ in a similar fashion



until we get to the bottom:



It is easy to see that the depth is $1 + \lg n$, and the total number of leaves is n. Each level takes a total of cn amounts of work, so the overall complexity of the merge sort algorithm is $\Theta(n \lg n)$.

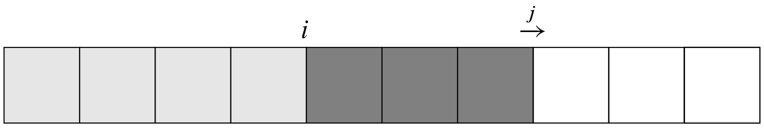


Figure 6: Illustration of QUICKSORT. i is the frontier of the smaller part, and j is the frontier of the larger part.

Our merge sort algorithm is not in place, and it takes $\Theta(n)$ auxiliary space. In-place implementation of merge sort does exist, but it is very complicated and not necessary performs well compared to the non in-place implementation, so it is not used much.

3.3 Quicksort

In quicksort, we select a *pivot* element, usually the first or the last element of the array, then put all elements that are less than the pivot to the left, and all elements that are greater than the pivot to the right. When then recursively sort the left part and the right part using the same method. When we finished the final array should become sorted. A somewhat nasty detail is how should we do this *in-place*, instead of creating new arrays each time. The implementation in Cormen et al. 2009 is Algorithm 3.3.1. The PARTITION procedure is meant to put elements of the array to the left and right of the pivot element in an in-place fashion. We imagine that we maintain smaller elements in the left part of the array and larger elements on the right part. i is the frontier of the smaller part while j is the frontier of the larger part. As j explores the whole array, if it discovers that $A[j]$ is smaller than the pivot, then we put it at the end of the larger part (in other words let it be the frontier of the smaller part). See Fig. 6 for an illustration.

Algorithm 3.3.1 Quicksort Algorithm

```
QUICKSORT($A$, $p$, $r$)
  **if** $q < r$ **then**
    $q$ = PARTITION($A$, $p$, $r$)
    QUICKSORT($A$, $p$, $q - 1$)
    QUICKSORT($A$, $q + 1$, $r$)

PARTITION($A$, $p$, $r$)
  $x = A[r]$
  $i = p - 1$
  **for** $j = p$ to $r - 1$ **do**
    **if** $A[j] \leq x$ **then**
      $i = i + 1$
      exchange $A[i]$ and $A[j]$
  exchange $A[i + 1]$ and $A[r]$
  **return** $i + 1$
```

Regarding the running time of quicksort, we have to make $\Theta(n)$ comparisons each time, and in the balanced case each time we reduce the array to two subarrays, so $T(n) = \Theta(n) + 2T(n/2)$. Thus the average running time is $\Theta(n \lg n)$. It is easy to see from the algorithm that if the array A is already sorted, then each time we would do $n - 1$ comparisons to make the problem reduce size by only 1 element, thus the worst case running time is $\Theta(n^2)$.

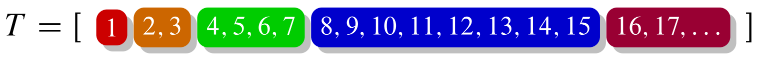


Figure 7: Heap T as an array. The numbers are indices. Elements that are on the same level are shown in the same color block.

3.4 Heapsort

(Binary) heaps are nearly complete binary trees with the additional *max-heap* or *min-heap* property. A complete binary tree of height h, denoted by T_h, is the binary tree such that all nodes except the leaves have two children. The zero level has one node, namely the root; the first level has 2 nodes, the second level has $2 \cdot 2 = 4$ nodes, the third level has $2 \cdot 4 = 8$ nodes...and the h level has 2^h nodes (all of them are leaves). The total number of nodes of such a complete binary tree is thus

$$1 + 2 + \cdots + 2^h = 2^{h+1} - 1.$$

A heap $T \subseteq T_h$ of height h is T_h with 0 or more leaves removed. Thus, the number of nodes $n(T)$ for a heap T with height h is bounded by

$$(2^h - 1) + 1 \leq n(T) < (2^{h+1} - 1) + 1.$$

So we can derive

$$\begin{gathered} 2^h \leq n(T) < 2^{h+1} \\ \Downarrow \\ h \leq \lg n(T) < h + 1 \\ \Downarrow \\ \lfloor \lg n(T) \rfloor = h. \end{gathered}$$

There are two types of heaps:

- A *max-heap* is a heap where each node is always larger or equal to its children;
- A *min-heap* is a heap where each node is always smaller or equal to its children.

We shall concentrate our analysis on max-heaps. The situation for min-heaps are completely similar. As noted in Cormen et al. 2009, we can implement heaps using *arrays*. Instead of viewing heaps as trees, we can also imagine heaps as in Fig. 7. From top to down, left to right, we index each element in the tree T as $1, 2, 3, \ldots$. Then we can see that for a node i, its left child is in position $2i$ and its right child is in $2i + 1$. Thus the parent of a node i is $\lfloor i/2 \rfloor$.

Given a node i, the MAX-HEAPIFY procedure looks at the "triangle△" $\langle i, i.\mathit{left}, i.\mathit{right} \rangle$ and see whether i is the largest. If it is small, then exchange it wit one of its children. Then after we put it down, we continue to look at the "triangle△" and do the same, until we have put the node i down to a suitable position. It is easy to see that the MAX-HEAPIFY procedure has running time $O(h) = O(\lg n)$. In Cormen et al. 2009, a variable $size$ is maintained so as to delimit elements from the heaps from elements that are removed and put at the end of the array.

```
MAX-HEAPIFY($T$, $i$)
    $l = \text{LEFT}(i)$; $r = \text{RIGHT}(i)$
    if $l \leq T.\mathit{size}$ and $T[l] > T[i]$ then
        $\mathit{largest} = l$
    else
        $\mathit{largest} = i$
    if $r \leq T.\mathit{size}$ and $T[r] > T[\mathit{largest}]$ then
        $\mathit{largest} = r$
    if $largest \neq i$ then
        exchange $T[i]$ with $T[largest]$
        MAX-HEAPIFY($T$, $largest$)
```

To build a max-heap, we can start at the bottom of the tree, go up, and put every small element we encountered down using MAX-HEAPIFY.

```
BUILD-MAX-HEAP($T$)
    $T.size = T.length$
    for $i = \lfloor T.length/2 \rfloor$ to 1 do
        MAX-HEAPIFY($T, i$)
```

At any *level* l (root has level 0 and leaves have level $l = h$), the number of nodes of a heap is at most 2^l. For a node of level l, $h' = h - l$ is the "height" of the node as used in Cormen's book. It is easy to see that for a complete binary tree T_h of height h, we have $2^l = \lceil n/2^{h-l+1} \rceil$ for any $l = 0, 1, \ldots, h$. Indeed

$$\frac{n}{2^{h-l+1}} = \frac{2^{h+1}-1}{2^{h-l+1}} = \frac{1}{2^{-l}} \frac{2^{h+1}-1}{2^{h+1}} = 2^l \left(1 - \frac{1}{2^{h+1}}\right) \quad \text{so that} \quad \left\lceil \frac{n}{2^{h-l+1}} \right\rceil = 2^l.$$

Thus for a heap, every height h' has at most $2^l = \lceil n/2^{h'+1} \rceil$ nodes. The cost of BUILD-MAX-HEAP is then

$$\sum_{h'=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h'+1}} \right\rceil O(h') = O\left(n \sum_{h'=0}^{\lfloor \lg n \rfloor} \frac{h'}{2^{h'}}\right) = O\left(n \sum_{h'=0}^{\infty} \frac{h'}{2^{h'}}\right) = O(n).$$

By the property of max-heaps, the root is the largest element of the heap. So to sort an array using heaps, we can first build a max-heap, then put the root at the end of the array, i.e. exchange the root with the last element, and then run MAX-HEAPIFY(T, 1), to put second largest element in the root. We can then repeat the above procedure until a two-element heap remains. The running time is $O(n \lg n)$.

Algorithm 3.4.1 Heapsort

```
HEAPSORT($T$)
    BUILD-MAX-HEAP($T$)
    for $i = T.length$ to 2 do
        exchange $T[1]$ with $T[i]$
        $T.size = T.size - 1$
        MAX-HEAPIFY($T$, 1)
```

3.4.1 Priority Queues

We can easily see that max-heaps can be used to implement priority queues. A max-priority queue is a queue, in which the order of the elements are determined by their key values. It is a data structure Q supporting the following operations:

- MAXIMUM(Q): return the first element of the queue.
- EXTRACT-MAX(Q): remove and return the first element of the queue.
- INSERT(Q, x): insert the element x into the queue.
- INCREASE-KEY(Q, x, k): increase the value of x to the new value k.

The implementation is as Algorithm 3.4.2. The running times of the four operations are

- Maximum(Q): $\Theta(1)$.
- Extract-Max(Q): $O(\lg n)$.
- Insert(Q, x): $O(\lg n)$.
- Increase-Key(Q, x, k): $O(\lg n)$.

Algorithm 3.4.2 Max-Priority Queue

```
Maximum($Q$)
    return $Q[1]$


Extract-Max($Q$)
    if $Q.size < 1$ then
        error “heap underflow”
    $max,\ Q[1] = Q[1],\ Q[size]$
    $Q.size = Q.size - 1$
    Max-Heapify($Q$, 1)
    return $max$


Insert($Q, k$)
    $Q.size = Q.size + 1$
    $Q[Q.size] = -\infty$
    Increase-Key($Q$, $Q.size, k$)


Increase-Key($Q, i, k$)
    if $k < Q[i]$ then
        error “new key is smaller than current key”
    $Q[i] = k$
    while $i > 1$ and $Q$[Parent($i$)] $< Q[i]$ do
        exchange $Q[i]$ with $Q$[Parent($i$)]
        $i$ = Parent($i$)
```

4 Graph Theory and Graph Algorithms

4.1 Basic Definitions and Properties

We use $G = (V, E)$ to denote a graph. Often we will assume that the graph is *simple*. This means it does not contain self-loops or multiple edges. What is the maximum number of edges $|E|$ possible for a simple graph? If every vertex $v \in V$ is connected to all other $|V| - 1$ vertices, then the total number of edges is $|V| \cdot (|V| - 1)/2$. Thus, we have $|E| \leq |V| \cdot (|V| - 1)/2$.

Bipartite Graphs $G = (V, E)$ is called *bipartite* if V is the union of two disjoint sets X and Y where every edge in E is of the form (x, y) with $x \in X$ and $y \in Y$. The matching problem can be solved in polynomial time. However, the matching problem for tripartite graph is NP-complete.

Vertex Colorings A vertex coloring of G is a map $f : V \to C$ from the set of vertices V to a set of colors C. A coloring is *proper* if and only if for each edge $(a, b) \in E$, we have $f(a) \neq f(b)$. The *chromatic number* $\mathcal{X}(G)$ is the minimum number of colors in a proper coloring of G. If we fix the $|C|$, then the problem of finding proper coloring is NP-complete.

Planar Graphs A graph is *planar* if there is no overlapping edge when draw it on a plane. The four color theorem says that four colors are enough for planar graphs for the proper coloring problem. However, the problem for a general graph is still NP-complete.

Subgraphs $G' = (V', E')$ is a *subgraph* of G if $V' \subset V$ and $E' \subset E$. G' is a *spanning subgraph* if $V' = V$. If $V' \subset V$, then the *subgraph induced by* V' is $G[V'] = (V', E')$ where $E' = \{(u, v) \in E \mid u, v \in V'\}$. If $E' \subset E$, then the *subgraph induced by* E' is $G[E'] = (V', E')$, where $V' = \{v \in V \mid \exists e \in E' \text{ such that } v \in e\}$.

Graph isomorphisms $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there is a bijection $f : V \to V'$ between vertices of G and G' such that u is adjacent to v if and only if $f(u)$ is adjacent to $f(v)$, i.e. $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. The problem of determining whether two graphs are isomorphic is NP.

Complete Graphs A *complete* graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge. We denote the complete graph of n vertices by K_n. A complete bipartite graph or biclique is a special kind of bipartite graph where every vertex of the first set is connected to every vertex of the second set.

Degrees The *degree* of a vertex v, denoted by $d(v)$, is the number of edges incident with v. In a simple graph the degree of any v is at most $|V| - 1$. We define

$$\delta(G) = \min_{v \in V} d(v) \quad \text{and} \quad \Delta(G) = \max_{v \in V} d(v).$$

Proposition 4.1.

$$\sum_{v \in V} d(v) = 2|E|.$$

Proof. Consider the incidence matrix representation of the graph. Each row v has $d(v)$ 1s. This implies that the total number of 1s in the incidence matrix is $\sum_v d(v)$. Each column e has two 1s. We can sum by columns and the total number of 1s is also $2|E|$. □

Proposition 4.2. *In any graph $G = (V, E)$, the number of vertices of odd degrees is even.*

Proof. Let $\mathbb{Z}/\mathbb{Z}_2 = \{\bar{0}, \bar{1}\}$ denote the group of order 2. Let's partition V into vertices of odd degrees $A = \{v \in V \mid \overline{d(v)} = \bar{1}\}$ and even degrees $B = \{v \in V \mid \overline{d(v)} = \bar{0}\}$. Then

$$\sum_{v \in A} d(v) + \sum_{v \in B} d(v) = 2|E| \quad \Rightarrow \quad \overline{\sum_{v \in A} d(v)} = \overline{2|E|} - \overline{\sum_{v \in B} d(v)} = \bar{0} - \bar{0} = \bar{0}.$$

Thus the number of terms must be even, i.e. $\overline{|A|} = \overline{0}$. □

Graph Representations The *adjacency matrix* of a graph $G = (V, E)$ is the $|V| \times |V|$ matrix A such that $A_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise. Note that A is symmetric, with 0 on the diagonals (for simple graph). A graph can also be represented by the *incidence matrix* M. It is the $|V| \times |E|$ matrix such that

$$M(v, e) = \begin{cases} 1 & v \in e; \\ 0 & \text{otherwise.} \end{cases}$$

Note that the sum of elements in a row is the degree of that vertex, while the sum in a column is always 2.

Proposition 4.3. *Let A be the adjacency matrix of a graph G. Then $A^k(s, t)$ is the number of walks of length k from $s \in V$ to $t \in V$.*

Proof. Let $N_k(v, w)$ be the number of walks from v to w with k edges, and let $N_k(v, w; u)$ be the number of walks from v to w with k edges whose second-to-last vertex is u. We prove by induction. The theorem is true for $k = 1$. Suppose it is true for k. We have

$$N_{k+1}(v, w) = \sum_{u \in V} N_k(v, w; u) = \sum_{u \in V} N_k(v, u) A(u, w) = \sum_{u \in V} A^k(v, u) A(u, w) = A^{k+1}(v, w).$$

□

Paths A *walk* is a list of vertices $w = (v_1, \dots, v_k)$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i < k$. A walk may contain loops. A walk is *closed* if $v_1 = v_k$. A *cycle* is a closed walk. A *path* is a walk in which the vertices are distinct. A *trail* is a walk in which all edges are distinct. The relation $a \sim b$ if there is a walk from a to b is an equivalent relation on the graph G. The equivalent classes are called *connected components*. Denote $\omega(G)$ the number of components of G. G is connected if $\omega(G) = 1$. The components $C_1, \dots, C_r$ of G induce connected subgraphs.

Theorem 4.4. *A graph is bipartite if and only if it has no odd cycle.*

Proof. Necessity. Let $G = X \cup Y$ be a bipartite graph. Every walk alternates between X and Y, so every return to the original starting set happens after an even number of steps. Hence G has no odd cycle.

Sufficiency. Let $G = (V, E)$ be a graph with no odd cycle, and suppose G is connected (if not then just apply the argument to each component). We shall construct a bipartition of G. Let $u \in V$ be any vertex in G, and for each $v \in V$, let $f(v)$ be the minimum length of a $u \rightsquigarrow v$ path. Since we assume G is connected, $f(v)$ is well-defined for each $v \in V$.

Let $X = \{v \in V : \overline{f(v)} = \overline{0}\}$ and $Y = \{v \in V : \overline{f(v)} = \overline{1}\}$. If X or Y contains an edge $e = vv'$, then there would be a closed odd walk using a shortest $u \rightsquigarrow v$ path, the edge vv', and the reverse of a shortest $u \rightsquigarrow v'$ path, since $\overline{0} + \overline{0} + \overline{1} = \overline{1}$ and $\overline{1} + \overline{1} + \overline{1} = \overline{1}$. See Fig. 8. Hence X and Y are independent sets. Also $X \cup Y = V$, so G is indeed a bipartite graph.

□

4.2 Trees

A graph with no cycle is *acyclic*. A *forest* is an acyclic graph. A *tree* is a connected acyclic graph. A *leaf* is a vertex of degree 1. A *spanning subgraph* of G is a subgraph with vertex set $V(G)$. A *spanning tree* is a spanning subgraph that is a tree.

Suppose $G = C_1 \cup C_2 \cup \dots \cup C_r$. Suppose $e = (u, v) \in E$ such that $u \in C_i, v \in C_j$. Let $\omega(G)$ denote the number of connected components of G.

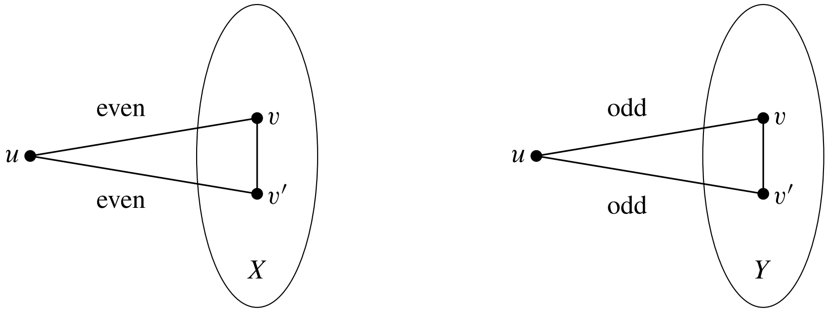


Figure 8: Illustration for the proof of Theorem 4.4. There should be no edge in X or Y, otherwise there would exist odd cycle in graph G.

- If $i = j$ then $\omega(G+e) = \omega(G)$
- If $i \neq j$ then $\omega(G+e) = \omega(G) - 1$.

Suppose $G = (V, E)$ is acyclic with components $C_1, C_2, \dots, C_r$. Let $n = |V|, e = (u, v) \in E$ where $u \in C_i$ and $v \in C_j$.

- If $i = j$ then $G + e$ contains a cycle
- If $i \neq j$ then $G + e$ is still acyclic and $\omega(G+e) = \omega(G) - 1$, and G has $n - k$ edges.

Take r random edges $E = \{e_1, \dots, e_r\}$. We claim $G_i = (V, \{e_1, \dots, e_i\})$ has $n - i$ components. We prove by induction. If $i = 0$ then G_0 has no edges, so it has n components. If $i > 0$, then G_{i+1} is acyclic. G_i remains acyclic. It follows that e_i joins vertices in distinct components of G_i. G_i has one less component of G_{i-1}. $r = n - k$.

If T is a tree with n vertices, then

1. it has $n - 1$ edges
2. it has at least 2 vertices of degree 1.

Proof. Let s be the number of vertices of degree 1 in T. For the tree

$$\sum_{v \in V} d(v) = 2(n-1).$$

Then

$$2n - 2 = \sum_{v \in V} d(v) = \sum_{v \in V_{-1}} d(v) + s \geq 2(n-s) + s$$

We have

$$2n - 2 \geq 2n - s \Rightarrow s \geq 2.$$

□

Theorem 4.5. *If $|V| = n$ and $|E| = n - 1$, then the following statements are equivalent:*

- *G is connected;*
- *G is acyclic;*
- *G is a tree.*

e is a *cut edge* if $\omega(G - e) > \omega(G)$, i.e. removing e would increase the connected components of G.

Theorem 4.6. *$e = (u, v)$ is a cut edge if and only if e is not on any cycle.*

Corollary 4.7. *A connected graph is a tree if and only if every edge is a cut edge.*

Corollary 4.8. *Every finite connected graph G contains a spanning tree.*

Proof. If there are no cycles, then we are done. If there is a cycle, delete one of its edges. Then the graph remains connected. Repeat this deleting process and the process must terminate because the number of edges is finite. On termination we have a spanning tree. □

4.3 Basic Graph Algorithms

4.3.1 Breath First Search

The BFS algorithm puts every neighbors (except for those already visited) of every vertex into the queue. Thus, it will start by exploring all neighbors of s, and then it will explore all neighbors of the first neighbors of s...

Algorithm 4.3.1 Breadth First Search

```
BFS($G, s$)
    $s.color$ = GRAY; $s.d = 0$; $s.\pi$ = NIL
    for each vertex $u \in V \setminus \{s\}$ do
        $u.color$ = WHITE; $u.d = \infty$; $u.\pi$ = NIL
    $Q = \varnothing$; ENQUEUE($Q, s$)
    while $Q \neq \varnothing$ do
        $v$ = DEQUEUE($Q$)
        for each $u \in Adj[v]$ do
            if $u.color$ == WHITE then
                $u.color$ = GRAY
                $u.d = v.d + 1$
                $u.\pi = v$
                ENQUEUE($Q, u$)
        $v.color$ = BLACK
```

The BFS algorithm enqueues every vertex at most once, and hence dequeue every vertex at most once. The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. Thus the total running time of BFS is $O(V + E)$.

4.4 Minimum Spanning Trees

Given a connected, undirected graph $G = (V, E)$ with weight $w : E \to \mathbb{R}$, we'd like to find an acyclic subset T of E that connects all the vertices (a spanning tree) whose weight

$$w(T) = \sum_{e \in T} w(e)$$

is minimal. How do we find all edges of a minimum spanning tree? The following theorem provides a mean. It says that if we have any subset $U \subseteq V$, then the edge with minimum weight connecting any $u \in U$ and any $v \in V \setminus U$

must be part of a minimum spanning tree. If it is not in the MST, then we can find another edge e' with larger weight connecting U and $V \setminus U$. Deleting e' and adding e yields a spanning tree with smaller total weight.

Theorem 4.9. *Let $U \subseteq V$ be any subset of vertices of G. Let e be the edge with the smallest weight $w(e)$ connecting U and $V \setminus U$, i.e. $e = \arg\min_{(u,v)}\{w(u,v) \mid u \in U \text{ and } v \in V \setminus U\}$. Then e is part of the minimum spanning tree.*

Proof. Suppose by contradiction T is a minimum spanning tree not containing $e = (u, v)$. Since T is a spanning tree, this implies that it contains a unique path between u and v. This path must contain an edge e' connecting U and $V \setminus U$ that is different from $e = (u, v)$. This path together with $e = (u, v)$ forms a cycle in G. $T + e - e'$ is another spanning tree with $w(T + e - e') < w(T)$, contradicting to the assumption that T is a minimum spanning tree. □

4.4.1 Prim's Algorithm

In light of Theorem 4.9, we see how we can find a minimum spanning tree: start with the source vertex $U_0 = \{s\}$ and find the minimum weight edge (s, u_1) that connects s. Let $U_1 = \{s, u_1\}$. Then for all edges connecting U_1 find the minimum edge with vertex $u_2 \in V \setminus U_1$. Let $U_2 = \{s, u_0, u_1\}$ and so on, until we get to $U_{|V|} = V$. In symbol

$$u_{n+1} = \arg\min\{w(u,v) \mid u \in U_n, v \in V \setminus U_n\} \quad \text{and} \quad U_{n+1} = U_n \cup \{u_{n+1}\}.$$

This is the Prim's algorithm. However, to efficiently implement this idea, we need a little tweak. Since we need to select the minimum-value vertex from $V \setminus U_n$ each time, we can use a min-priority queue to do this.

Algorithm 4.4.1 Prim's Algorithm

```
PRIM($G, w, s$)
  for each $v \in V$ do
      $v.key = \infty$, $v.\pi = $ NIL
  $s.key = 0$, $Q = V$
  while $Q \neq \emptyset$ do
      $u = $ EXTRACT-MIN($Q$)
      for each $v \in Adj[u]$ do
          if $v \in Q$ and $w(u,v) < v.key$ then
              $v.key = w(u,v)$
              $v.\pi = u$
```

The running time of Prim's algorithm depends on how we implement the min-priority queue Q. If we implement Q as a binary min-heap, we can use the BUILD-MIN-HEAP procedure to perform the initializations in $O(V)$ time. The body of the **while** loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$. The **for** loop executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, we can implement the test for membership in Q in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q. The assignment in the second-to-last line involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$.

4.4.2 Kruskal's Algorithm

Kruskal's algorithm (Algorithm 4.4.2) makes use of Theorem 4.9 in a different way. The idea is to first sort the edges by weight in ascending order. Then the minimal edge e_1 must be in some minimum spanning tree: just take U in

Theorem 4.9 to be any of the two endpoints of e_1. The second smallest edge e_2 is also in some MST: we can take U to be any of the two endpoints of e_2 that is not in e_1. In general, we scan the sorted list of E and include e into our minimum spanning tree as long as it does not connect two vertices that are already in the same component. The algorithm mainly spends time sorting the edges, so the total running time is $O(E \lg E)$. Since $|E| \le |V|^2$, we have $\lg E = O(\lg V)$, so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$, which is the same as Prim's algorithm.

Algorithm 4.4.2 Kruskal's Algorithm

```
KRUSKAL($G, w$)
    $A = \varnothing$
    for each vertex $v \in V$ do
        MAKE-SET($v$)
    E = SORT($E, w$, ascending = TRUE)
    for each edge $(u, v) \in E$ do
        if FIND-SET($u$) $\neq$ FIND-SET($v$) then
            $A = A \cup \{(u, v)\}$
            UNION($u, v$)
    return $A$
```

4.5 Single-Source Shortest Paths

The single-source shortest path problem is concerned with finding the shortest path from one source vertex s to all other reachable vertices. Without loss of generality, in the following we assume that our directed graph $G = (V, E)$ at consideration is connected. The weight of a path $p = \langle s, v_0, v_1, \dots, v_k \rangle$ is the sum of the weight of all the edges in p:

$$\omega(p) = w(s, v_0) + w(v_0, v_1) + \cdots + w(v_{k-1}, v_k).$$

We are interested in finding the function

$$d^* : V \to \mathbb{R},$$

where for each $v \in V$, $d^*(v)$ is the weight of the shortest path from s to v, i.e. $d^*(v) = \min\{w(p) \mid p : s \rightsquigarrow v\}$. Below we shall construct the function

$$d : V \to \mathbb{R}$$

to approximate d^* and gradually improve values of d towards d^*. We use $\pi : V \to V$ to denote the predecessor function. We shall also construct optimal π along the way.

First we will set $d(v) = \infty$ for all $v \in V \setminus \{s\}$ and $d(s) = 0$ for the source. This way our algorithms can begin with the source s. This is the INITIALIZE-SINGLE-SOURCE procedure.

```
INITIALIZE-SINGLE-SOURCE($G, s$)
    for each vertex $v \in V$ do
        $d(v) = \infty$
        $\pi(v) =$ NIL
    $d(s) = 0$
```

The next question is, how do we improve d? Well, for any edge $(u, v) \in E$, we can ask whether going from s to v through u (i.e. try this different path) can reduce the current estimate $d(v)$. Namely, we can let

$$d(v) = \min\left\{d(v),\ d(u) + w(u, v)\right\}. \tag{2}$$

This is called the “relaxation” procedure.

```
RELAX($u$, $v$, $w$)
    if $d(v) > d(u) + w(u, v)$ then
        $d(v) = d(u) + w(u, v)$
        $\pi(v) = u$
```

4.5.1 The Bellman-Ford Algorithm

The Bellman-Ford algorithm is simple: for each edge $e \in E$, relax e, and do this for $|V| - 1$ times.

Algorithm 4.5.1 Bellman-Ford Algorithm

```
BELLMAN-FORD($G$,$w$, $s$)
    INITIALIZE-SINGLE-SOURCE($G$, $s$)
    for $i = 1$ to $|V| - 1$ do
        for each edge $(u, v) \in E$ do
            RELAX($u$, $v$, $w$)
    for each edge $(u, v) \in E$ do          ☞ detect negative weight cycles
        if $d(v) > d(u) + w(u, v)$ then
            return FALSE
    return TRUE
```

The real question is, why does it work?

Pick *any* vertex $v_k \in V$, and consider an *optimal* path $p = \langle s, v_0, v_1, \ldots, v_k \rangle$ leading to v_k. The crucial point is that, $\langle s, v_0, \ldots, v_j \rangle$ must be the optimal path leading to v_j for *any* vertex v_j along p. For if we can find a shorter path leading to v_j, then we can substitute $\langle s, v_0, \ldots, v_j \rangle$ with this shorter path, so that we get a shorter path leading to v_k, contradicting to optimality of p. This implies

$$\begin{aligned} w(p) &= \underbrace{w(s, v_0)} + w(v_0, v_1) + w(v_1, v_2) + \cdots + w(v_{k-1}, v_k) \\ &= \underbrace{d^*(v_0) + w(v_0, v_1)} + w(v_1, v_2) + \cdots + w(v_{k-1}, v_k) \\ &= \underbrace{d^*(v_1) + w(v_1, v_2)} + \cdots + w(v_{k-1}, v_k) \\ &= d^*(v_2) + \cdots + w(v_{k-1}, v_k) \\ &= \cdots \\ &= d^*(v_k). \end{aligned}$$

Remember that we are assuming $p = \langle s, v_0, v_1, \ldots, v_k \rangle$ is *optimal*, and we derived the conclusion that each subpath $s \to v_0 \to \cdots \to v_j$ must also be optimal for $j = 0, \ldots, k - 1$. During the first pass, we relaxed $(s, v_0) \in E$ among all edges in E. We don’t know which is (s, v_0), but since we relaxed them all, we are sure that we must relaxed this particular edge. After the first pass we have $d(v_0) = d^*(v_0)$. Similarly, during the second pass, we relaxed $(v_0, v_1) \in E$ among all edges in E. After this pass we have $d(v_1) = d^*(v_0) + w(v_0, v_1) = d^*(v_1)$,...and so on.

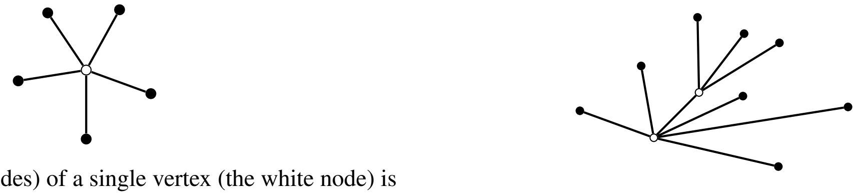
The running time of the Bellman-Ford algorithm is clearly $O(VE)$.

4.5.2 Dijkstra’s Algorithm

Dijkstra’s algorithm works on graphs with non-negative weights. It is actually quite similar to Prim’s algorithm. See Algorithm 4.5.2.

Algorithm 4.5.2 Dijkstra's Algorithm

```
DIJKSTRA($G$,$w$,$s$)
    INITIALIZE-SINGLE-SOURCE($G, s$)
    $S = \varnothing, Q = V$
    while $Q \neq \varnothing$ do
        $u =$ EXTRACT-MIN($Q$), $S = S \cup \{u\}$
        for each vertex $v \in Adj[u]$ do
            RELAX($u, v, w$)
```



(a) Neighbors (black nodes) of a single vertex (the white node) is the set of vertices adjacent to it.

(b) Neighbors (black nodes) of a subset (white nodes) of V.

Figure 9: Neighbors of subsets of vertices

Again, why does it work? To facilitate our analysis, for any single vertex $u \in V$ we define the *neighbors* of u, denoted by $\Gamma(u)$, to be $Adj[u]$, and we define the neighbors for any $U = \{u_1, \ldots, u_n\} \subseteq V$, $\Gamma(U) = \Gamma(u_1, \ldots, u_n)$, to be $\bigcup_{u \in U} Adj[u] \setminus U$. See Fig. 9 for illustrations. Note that at each **while** loop $u =$ EXTRACT-MIN$(Q) =$ EXTRACT-MIN$(\Gamma(U))$, since all other vertices have infinity values.

The algorithm begins by setting $d(v) = \infty$ for $v \in V \setminus \{s\}$ and $d(s) = 0$. Start from the source s, and set $d(v) = w(s, v)$ for each $v \in \Gamma(s)$. At this point it may be that $d(v) = d^*(v)$ for several v that are neighbors of s. We don't know how many of them become optimal at this first pass, but one thing we do know is that at least for

$$u_1 = \arg\min\{d(v) \mid v \in \Gamma(s)\}$$

it *must* be that $d(u_1) = d^*(u_1)$. Any *other* path from s to u_1 must pass $\Gamma(s)$, and since $d(v) \geq d(u_1)$ for all $v \in \Gamma(s)$, this can only increase the cost. Next, let $S_1 = \{s, u_1\}$ and relax all vertices in $\Gamma(u_1)$. We assert for

$$u_2 = \arg\min\{d(v) \mid v \in \Gamma(s, u_1)\}$$

we have $d(u_2) = d^*(u_2)$. The key observation is that *any* path p from s to u_2 *must* pass some vertex in $\Gamma(s, u_1) = \Gamma(s) \cup \Gamma(u_1)$:

(1) it would either pass s and u_1 to $u_2 \in \Gamma(s, u_1)$;

(2) or pass through some vertex $v \in \Gamma(u_1)$ to u_2;

(3) or otherwise pass through some vertex $v \in \Gamma(s)$.

We then have $w(p) \geq d(v) \geq d(u_2)$ for all $v \in \Gamma(s) \cup \Gamma(u_1)$, so going through v can only increase the cost. Since $w(p) \geq d(u_2)$ for any $p : s \rightsquigarrow u_2$, we have established that $d(u_2) = d^*(u_2)$. Note where relaxation of u_1 enters into the picture: after relaxation of u_1, each $d(v)$ for $v \in \Gamma(s, u_1)$ is the optimal value on the subset $\{p : s \rightsquigarrow v \mid p \cap S_1 \neq \varnothing\}$.

In general, after we have identified the vertices with optimal value $S_n = \{s, u_1, \ldots, u_n\}$ and relaxed u_n, we can identify the next optimal vertex by

$$u_{n+1} = \arg\min\{d(v) \mid v \in \Gamma(S_n)\}.$$

$d(u_{n+1})$ is then optimal, since there is some $v \in \mathcal{N}(S_n)$ on any path $p : s \rightsquigarrow u_{n+1}$, so $w(p) \geq d(v) \geq d(u_{n+1})$. We are able to make this statement because we have made the assumption that the edge weights are all non-negative, so the value of the path is no less than the value of any vertex on the path.

Running time Dijkstra's algorithm calls three min-priority queue operations: INSERT (implicit in "$Q = V$"), EXTRACT-MIN and DECREASE-KEY (implicit in RELAX). The algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each vertex $u \in V$ is added to set S exactly once, each edge in the adjacency list $Adj[u]$ is examined in the **for** loop exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, this **for** loop iterates a total of $|E|$ times, thus the algorithm calls DECREASE-KEY at most $|E|$ times overall.

The running time of Dijkstra's algorithm depends on how we implement the min-priority queue. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$. We simply store $v.d$ in the vth entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN operation takes $O(V)$ time (since we have to search through the entire array), for a total time of $O(V^2 + E) = O(V^2)$.

If the graph is sufficiently sparse–in particular, $E = o(V^2/\lg V)$–we can improve the algorithm by implementing the min-priority queue with a binary min- heap. Each EXTRACT-MIN operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. Each DECREASE-KEY operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E)\lg V)$, which is $O(E \lg V)$ if all vertices are reachable from the source. This running time improves upon the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.

Relationship with BFS and Prim's algorithm Dijkstra's algorithm resembles both breadth-first search (Section 4.3.1) and Prim's algorithm for computing minimum spanning trees (Section 4.4.1). It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so do black vertices in a breadth-first search have their correct breadth-first distances. Dijkstra's algorithm is like Prim's algorithm in that both algorithms use a min-priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in Prim's algorithm), add this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

4.6 All-Pairs Shortest Paths

In this subsection, we discuss algorithms for finding all-pairs shortest paths in directed, weighted graphs. For convenience we assume that the vertices are numbered $1, 2, \ldots, |V| = n$. Our input is the $n \times n$ weight matrix $W = (w_{ij})$ where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i, j) & \text{if } (i, j) \in E \\ \infty & \text{if } (i, j) \notin E. \end{cases}$$

Our goal is to compute the function $d^* : V \times V \to \mathbb{R}$, or equivalently the matrix $D^* = (d^*_{ij})$ where d^*_{ij} is the optimal value between vertex i and j. We shall use some $d : V \times V \to \mathbb{R}$ to approximate d^*. To keep track of the optimal paths we shall also compute the predecessor matrix $\Pi = (\pi_{ij})$, where $\pi_{ij} = \text{NIL}$ if $i = j$ or there is no path from i to j, and otherwise π_{ij} is the predecessor of j on some shortest path from i.

As a first thought, we may try to run a single-source shortest path algorithm $|V|$ times to find the shortest path between each vertex i and j. If the weights are non-negative, then we can use Dijkstra's algorithm, which gives us a running time of $O(V^3)$ if we use linear-array implementation of the min-priority queue, and $O(VE \lg V)$ if we use the min-heap implementation. If there are negative weights, then we have to use the Bellman-Ford algorithm, which gives us a running time of $O(V^2E)$. On dense graphs this is $O(V^4)$. Here we investigate whether we can have better methods.

4.6.1 Matrix Multiplication

How do we tackle the problem of all-pairs shortest path? Let's fix i and j and consider the optimal path $p : i \rightsquigarrow j$ whose length is at most m. We use $d^*_m(i, j)$ to denote the optimal value for $p : i \rightsquigarrow j$ that has length most m. As before, the crucial observation is that, since $p : i \rightsquigarrow j$ is optimal, for any other vertex k on p the path $p' : i \rightsquigarrow k$ is the optimal path between vertex i and k, where p' has length at most $m - 1$. In particular, for j's predecessor v we should have $d(i, v) = d^*_{m-1}(i, v)$ along p. So if we have the data $d^*_{m-1}(i, v)$ for all vertices $v \in V$, then

$$d^*_m(i, j) = \min_{v\in V} \left\{d^*_{m-1}(i, v) + w(v, j)\right\}.$$

Similarly, to calculate $d^*_{m-1}(i, v)$ for any $v \in V$ we do

$$d^*_{m-1}(i, v) = \min_{x\in V} \left\{d^*_{m-2}(i, x) + w(x, v)\right\}$$

and so on. This leads us all the way back to $d^*_0(i, v)$, which we define as

$$d^*_0(i, v) = \begin{cases} 0 & i = v \\ \infty & i \neq v. \end{cases}$$

Example 4.10. *For $m = 1$, $d^*_1(i, v) = \min_{x\in V}\{d^*_0(i, x) + w(x, v)\} = w(i, v)$, and for $m = 2$*

$$\begin{aligned} d^*_2(i, v) &= \min_{x\in V}\{d^*_1(i, x) + w(x, v)\} = \min_{x\in V}\{w(i, x) + w(x, v)\} \\ &= \begin{cases} w(i, v) & \text{if } x^* = i \\ w(i, x^*) + w(x^*, v) & \text{if } x^* \neq i. \end{cases} \end{aligned}$$

To calculate $d^*_3(i, v)$, we just need to consider $d^*_2(i, x)$ and $w(x, v)$ etc. We already stored all the information about path with length at most 2 in d^*_2, so we do not need to recalculate them again.

So, given $d^*_0(i, j)$, we are able to compute $d^*_1(i, j)$, and then $d^*_2(i, j)$, $d^*_3(i, j)$, . . . until $d^*_{n-1}(i, j) = d^*(i, j)$. At each step we can do this for all $i, j \in V$, so in matrix notation, once we have D^*_0, we can compute $D^*_1 = W$, then D^*_2, then D^*_3, . . . until $D^*_{n-1} = D^*$.

Algorithm 4.6.1 computes $D^*_m = \left(d^*_m(i, j)\right)$ from $D^*_{m-1} = \left(d^*_{m-1}(i, j)\right)$ for all $i, j \in V$. We note that the procedure is very similar to matrix multiplication, where we have $+$ in place of min and $\cdot$ in place of $+$, i.e. it would be $D^*_m(i, j) = D^*_m(i, j) + D^*_{m-1}(i, v) \cdot W(v, j)$. As in Cormen et al. 2009, we can borrow the notation for matrix multiplication and write EXTEND-SHORTEST-PATHS(D^*_{m-1}, W) as $D^*_{m-1} \cdot W$.

We can do EXTEND-SHORTEST-PATHS $n - 1$ times to solve the problem of all-pairs shortest paths. In matrix notations, this is W, $W \cdot W$, $W \cdot W \cdot W$ and so on, until W^{n-1}.

There are three **for** loops in Algorithm 4.6.1, and counting the **for** loop in Algorithm 4.6.2, the total running time of Algorithm 4.6.2 is $O(n^4)$.

In fact, we can do better: since EXTEND-SHORTEST-PATHS is associative, we don't have to "multiply" once at a time: we can "multiply" the output of each run with itself as W^2, W^4, ... This gives us a running time of $\Theta(n^3 \lg n)$.

We can view the dynamic programming strategy as first restricting ourselves to smaller spaces where it is relatively easy to search for the (approximate) solutions, and then gradually enlarge the spaces (relax the constraints). In

Algorithm 4.6.1 Extend Shortest Paths

```
Extend-Shortest-Paths($D^*_{m-1}$,$W$)
    $n$ = number of rows in $D^*_{m-1}$
    let $D^*_m$ be a new $n \times n$ matrix
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            $D^*_m(i, j) = \infty$
            for $v = 1$ to $n$ do
                $D^*_m(i, j) = \min\{D^*_m(i, j), D^*_{m-1}(i, v) + W(v, j)\}$
    return $D^*_m$
```

Algorithm 4.6.2 All-Pairs Shortest Paths - Slow Version

```
Slow-All-Pairs-Shortest-Paths($W$)
    $n$ = number of rows in $W$
    $D^*_1 = W$
    for $m = 2$ to $n - 1$ do
        let $D^*_m$ be a new $n \times n$ matrix
        $D^*_m$ = Extend-Shortest-Paths($D^*_{m-1}$, $W$)
    return $D^*_{n-1}$
```

Algorithm 4.6.3 All-Pairs Shortest Paths - Faster Version

```
Faster-All-Pairs-Shortest-Paths($W$)
    $n$ = number of rows in $W$
    $D^*_1 = W, \quad m = 1$
    while $m < n - 1$ do
        let $D^*_{2m}$ be a new $n \times n$ matrix
        $D^*_{2m}$ = Extend-Shortest-Paths($D^*_m$, $D^*_m$)
        $m = 2m$
    return $D^*_m$
```

Section 4.6.1, for any pair i and j we approximate the optimal path $p^* : i \rightsquigarrow j$ by the sequence $p_0^*, p_1^*, p_2^*, \ldots$ until $p_{n-1}^* = p^*$. These approximations are not independent of each other. There is a dependence of problems on smaller subproblems, so that previous approximations can help in finding the next approximations.

4.6.2 The Floyd-Warshall Algorithm

The Floyd-Warshall algorithm (Algorithm 4.6.4) uses a different strategy. Rather than approximate D^* by D_m^* (shortest paths that use only m or less edges), $m = 0, \ldots, n-1$, It approximates D^* by shortest paths that use only the first $\{1, 2, \ldots, k\}$ *vertices*, and let k goes from 0 to n to get better approximations. Let $p : i \rightsquigarrow j$ is such a shortest path between i and j, and denote the value of the path by $d_k^*(i,j)$. Either k is not in p, so that $d_k^*(i,j) = d_{k-1}^*(i,j)$, or k is in p, for which $d_k^*(i,j) = d_{k-1}^*(i,k) + d_{k-1}^*(k,j)$, i.e. the subpath from i to k must be optimal within $\mathcal{P}_k(i,k) = \{p : i \rightsquigarrow k \mid len(p) \le k-1\}$ and the subpath from k to j must be optimal within $\mathcal{P}_k(k,j) = \{p : k \rightsquigarrow j \mid len(p) \le k-1\}$, and the value of $d_k^*(i,j)$ is the sum of the values of the two subpaths. In summary

$$d_k^*(i,j) = \begin{cases} w_{ij} & k = 0 \\ \min\left\{d_{k-1}^*(i,j),\ d_{k-1}^*(i,k) + d_{k-1}^*(k,j)\right\} & k \ge 1. \end{cases}$$

Algorithm 4.6.4 Floyd-Warshall Algorithm

```
FLOYD-WARSHALL($W$)
    $n$ = number of rows in $W$
    $D_0^* = W$
    for $k = 1$ to $n$ do
        let $D_k^* = \left(d_k^*(i,j)\right)$ be a new $n \times n$ matrix
        for $i = 1$ to $n$ do
            for $j = 1$ to $n$ do
                $d_k^*(i,j) = \min\left\{d_{k-1}^*(i,j),\ d_{k-1}^*(i,k) + d_{k-1}^*(k,j)\right\}$
    return $D_n^*$
```

Since there are three **for** loops in Algorithm 4.6.4, the running time is $\Theta(n^3)$.

To obtain the shortest paths, we can compute the matrix Π_0, Π_1, ..., $\Pi_n = \Pi$ along the way of computing D^*, where π_{ij} is the predecessor of the vertex j on a shortest path from i to j. $\pi_k(i,j) \in \Pi_k$ is the the predecessor of the vertex j on a shortest path from i to j with the constraint that all intermediate vertices are in $\{1, \ldots, k\}$. For $k = 0$ we define

$$\pi_0(i,j) = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } (i,j) \notin E \\ i & \text{if } (i,j) \in E \end{cases}$$

For $k \ge 1$, we observe that for the path $i \rightsquigarrow j$, if k is actually not in the path, then $\pi_k(i,j) = \pi_{k-1}(i,j)$. If k is indeed in the path, then $\pi_k(i,j) = \pi_{k-1}(k,j)$.

4.6.3 Transitive Closure of a Directed Graph

Definition 4.11 (Transitive Closure)**.** *Given a directed graph $G = (V, E)$ with vertex $V = \{1, 2, \ldots, n\}$, the* transitive closure *of G is the graph $\overline{G} = (V, \overline{E})$ where for all $i, j \in V$ we put (i,j) in $\overline{E}$ if in G there is a path from i to j.*

We can compute the transitive closure using a procedure that is similar to Floyd-Warshall algorithm. Define $t_k(i,j)$ to be 1 if there is a path from i to j with all intermediate vertices in $\{1, \ldots, k\}$, and 0 otherwise. Our goal is to compute

$T_n = (T_n(i,j))$, and put (i,j) in $\overline{E}$ if $t_n(i,j) = 1$. For $k = 0$ we define

$$t_0(i,j) = \begin{cases} 0 & (i,j) \notin E \\ 1 & (i,j) \in E \end{cases}$$

and we can compute $t_k(i,j)$ as

$$t_k(i,j) = t_{k-1}(i,j) \vee [t_{k-1}(i,k) \wedge t_{k-1}(k,j)].$$

```
TRANSITIVE-CLOSURE($G$)
    let $T_0 = (t_0(i,j))$ be a new $n \times n$ matrix
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            if $(i,j) \in E$ then
                $t_0(i,j) = 1$
            else
                $t_0(i,j) = 0$
    for $k = 1$ to $n$ do
        let $T_k = (t_k(i,j))$ be a new $n \times n$ matrix
        for $i = 1$ to $n$ do
            for $j = 1$ to $n$ do
                $t_k(i,j) = t_{k-1}(i,j) \vee [t_{k-1}(i,k) \wedge t_{k-1}(k,j)]$
    return $T_n$
```

The algorithm runs in $\Theta(n^3)$ time.

4.6.4 Johnson's Algorithm

Johnson's algorithm finds shortest paths between all pairs in $O(V^2 \lg V + VE)$ time. For sparse graphs, it is asymptotically faster than either repeated squaring of matrices or the Floyd-Warshall algorithm. The algorithm either returns a matrix of shortest-path weights for all pairs of vertices or reports that the input graph contains a negative-weight cycle. Its uses as subroutines both Dijkstra's algorithm and the Bellman-Ford algorithm.

Given a graph $G = (V, E)$ with weight $w : E \to \mathbb{R}$, The algorithm works as follows:

1. First add a new vertex s to the graph $G = (V, E)$, and $|V|$ new edges connecting s to each $v \in V$. Set the weight of $(s, v) \in E$ to be 0. Then run the Bellman-Ford algorithm with s as the source vertex, to find the function $h : V \to \mathbb{R}$ that gives shortest path weight from s to each $v \in V$.
2. *Reweight* each edge (u, v) as $\widehat{w}(u,v) = w(u,v) + h(u) - h(v)$. Since $h(v)$ is the optimal value obtained by the Bellman-Ford algorithm, we have $h(v) \leq h(u) + w(u,v)$. Consequently $\widehat{w}(u,v) \geq 0$ for any edge $(u,v) \in E$. Also, under this new weight, the shortest path between any two vertices does not change.
3. Remove s and added edges, and for each vertex $v \in V$ run Dijkstra's algorithm with source v on the reweighted graph.
4. Finally, subtract $(h(u) - h(v))$ from computed $d^*(i,j)$ for every edge $(u,v) \in E$ to get the original optimal value.

To verify the claim in Item 2, consider any path $p = \langle x, v_1, \ldots, v_n, y \rangle$ between x and y. The new weight of the path is

$$\begin{aligned}
\widehat{w}(p) &= \widehat{w}(x, v_1) + \widehat{w}(v_1, v_2) + \cdots + \widehat{w}(v_n, y) \\
&= [w(x, v_1) + h(x) - h(v_1)] + [w(v_1, v_2) + h(v_1) - h(v_2)] + \cdots + [w(v_n, y) + h(v_n) - h(y)] \\
&= w(p) + h(x) - h(v_1) + h(v_1) - h(v_2) + \cdots + h(v_n) - h(y) \\
&= w(p) + h(x) - h(y),
\end{aligned}$$

so

$$\arg\min_p \widehat{w}(p) = \arg\min_p w(p).$$

If we implement the min-priority queue in Dijkstra's algorithm by a Fibonacci heap, Johnson's algorithm runs in $O(V^2 \lg V + VE)$ time. The simpler binary min-heap implementation yields a running time of $O(VE \lg V)$, which is still asymptotically faster than the Floyd-Warshall algorithm if the graph is sparse.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Table 2: An example of a price table for the rod cutting problem.

5 Dynamic Programming

Dynamic programming is a powerful technique for solving optimization problems. It relies on a simple principle, the principle of optimality: if a path from a to b is an optimal path, then for any point c on the path from a to b, the sub-path from c to b must be the optimal path from c to b. Thus, we can construct solutions to bigger problems from smaller problems. We can first work out the small problems (e.g. optimal path from c to b), store them, and then derive the values for bigger problems from smaller one. Instead of deriving the optimal value to the original solution, it constructs a whole value function $f(n)$ for *every* input size n.

5.1 Rod Cutting

The rod cutting problem is: given a rod a length n and a price table for each length $1, \ldots, n-1, n$, how to cut the rod so as to maximize the total revenue? An example of a price table is Table 2.

There are 2^{n-1} different ways of cutting a rod of length n: at every cut point in $\{1, \ldots, n-1\}$ we can decide whether or not to give a cut. At first the problem seems to be untrackable, but dynamic programming can come to rescue. The key observation is that, after we make the first cut, we divide the rod into two pieces. The value of this cut is the price of the first piece plus the value of the second piece. Let $f(n)$ denote the optimal value function. Then

$$f(n) = \max_{1 \le i \le n} \{p_i + f(n-i)\}.$$

This gives us a practical way for finding the optimal value function $f(n)$: since the value for larger n depends on the value for smaller n, we can first work out $f(n)$ for small n, store them in a table, and then use these values to calculate $f(n)$ for larger n. For example, the first few values for $f(n)$ is shown in Table 3. In calculating $f(n)$, we make use of all entries in the array $[0, f(1), \ldots, f(n-1)]$, so the running time of the algorithm is $\Theta(n^2)$.

```
CUT-ROD($p, n$)
    let $f[0..n]$ be a new array
    $f[0] = 0$
    for $j = 1$ to $n$ do
        $v = -\infty$
        for $i = 1$ to $j$ do
            $v = \max(v, p[i] + f[j-i])$
        $f[j] = v$
    return $f[n]$
```

To keep track of the optimal solution, in the second **for** loop we can record $i = \arg\max_{1 \le i \le n}\{p_i + f(n-i)\}$ for each $n = 1, \ldots, n$ in an array s, then to print the optimal cut we call

```
while $n > 0$ do
    print $s[n]$
    $n = n - s[n]$.
```

n	0	1	2	3	$\cdots$
$f(n)$	0	p_1	$\max\{p_1 + p_1, p_2\}$	$\max\{p_1 + f(2), p_2 + f(1), p_3\}$	$\cdots$

Table 3: The optimal value function $f(n)$ for the rod cutting problem.

5.2 Longest Increasing Subsequence

In longest increasing subsequence (LIS) problems, we are given a finite sequence of integers A and we want to find the length of the longest subsequence of A such that all elements of the subsequence are in increasing order. Let's use some examples to explain the terms:

- A (finite) *sequence* $A = (a_1, a_2, \dots, a_n)$ is just an array. Example: $A = (10, 22, 9, 33, 21, 50, 41, 60)$.
- The term *subsequence* is the same as in mathematics: a subsequence of A is (a_{i_k}) for some $0 < i_1 < i_2 < \cdots < n$. Examples of subsequences of A are (10), $(10, 22)$, $(10, 9, 33)$, $(33, 21, 60)$, and $(50, 60)$.
- An *increasing subsequence* of A is a subsequence (a_{i_k}) of A such that $a_{i_1} < a_{i_2} < \cdots$. Examples are (10), $(9, 33, 41)$, $(33, 41, 60)$, $(33, 50, 60)$, (41) etc.
- A *longest increasing subsequence* of A is the largest increasing subsequence of A. Note that LIS may not be unique. For example, A has two longest increasing subsequence $(10, 22, 33, 50, 60)$ and $(10, 22, 33, 41, 60)$.

Let $f(a_1, \dots, a_n)$ denote the optimal value function that takes an array as input and returns the length of a LIS of the array. The dynamic programming relationship is

$$f(a_1, \dots, a_j) = 1 + \max\{f(a_1, \dots, a_i) \mid 1 \le i < j,\ a_i < a_j\}. \tag{3}$$

Namely, if a subsequence is optimal, then after we removed the last element, the remaining subsequence should be optimal as well. We define $\max\{\} = 0$. We can start by working out small problems: for $n = 1$, $f(a_1) = 1$. For $n = 2$, if $a_1 < a_2$ then $f(a_1, a_2) = 2$ but if $a_1 > a_2$ then $f(a_1, a_2) = 1$. For $n = 3$, if $a_2 < a_3$, then $f(a_1, a_2, a_3) = 1 + f(a_1, a_2)$, etc. At position j, we look at all previous elements $\{a_1, \dots, a_{j-1}\}$ in the array, and find among those that are smaller than a_j the one with largest value. We build the value table this way from left to right, until n. See Table 4 for an example. It is clear that the running time of this algorithm is $O(n^2)$. Note that there is *constraint* in the relationship, namely we have to search among elements that are smaller than a_j. Otherwise if $a_i > a_j$, the optimal subsequence up to a_i concatenated by a_j would not constitute a feasible solution. Thus, the constraint has the effect of discarding some points in the search space.

```
LIS($a$)
    $n = len(a)$
    **if** $n = 0$ **then**
        **return** 0
    $f = [1\,..\,1]$
    **for** $j = 2$ to $n$ **do**
        $v = 0$
        **for** $i = 1$ to $j$ **do**
            **if** $a[i] < a[j]$ **then**
                $v = \max(v, f[i])$
        $f[j] = v + 1$
    **return** $\max(f)$
```

index			i			j		
A	10	22	9	33	21	50	41	60
f	1	2	1	3	2	4	4	5

Table 4: An example of value table for the longest increasing subsequence problem.

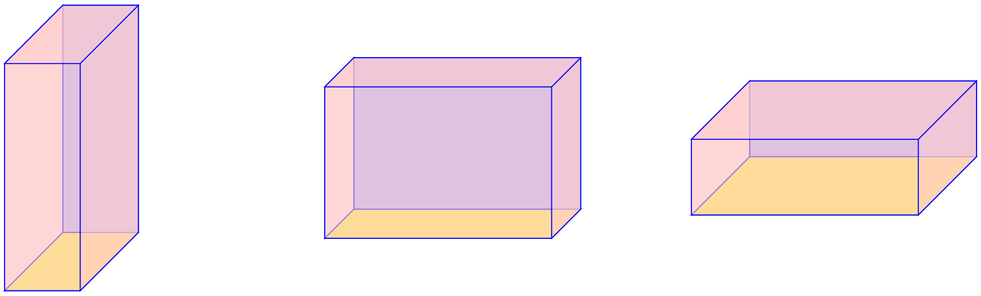


Figure 10: Three rotations of a box.

5.3 Box Stacking

The statement of the box stacking problem is as follows: you are given n types of boxes, where each box is defined by it's height, width and depth. You have an infinite supply of every type of box and you can rotate a box any way you like (exchanging height, width and depth). Two boxes can be stacked on top of each other if the width of the lower box is *strictly* smaller than the width of the upper box and the depth of the lower box is *strictly* smaller than the depth of the upper box (this prevents one from stacking infinitely many boxes of the same type) Write a dynamic programming algorithm that determines the maximal height you can achieve when stacking boxes, given the dimensions of the boxes.

How do we solve the box stacking problem? First note that, for a rectangular box, it can have three different base areas (see Fig. 10):

$$width \times height, \quad width \times depth, \quad height \times depth.$$

We have infinite supply of each type of the n boxes, but it suffices to have 3 of them for each of the boxes. Thus we consider stacking of the $3n$ boxes. A box with large base area cannot be stacked onto a box with small base area, so we may want to consider boxes with larger base areas, before we consider how to stack small boxes on other ones.

Let's put all $3n$ boxes in an array B, and *sort* the array B in *decreasing order* by its elements' base areas. We suppose we have three functions $h : B \to \mathbb{R}_+$, $w : B \to \mathbb{R}_+$ and $d : B \to \mathbb{R}_+$ that can return us the height, width and depth of a box $b \in B$. By width and depth of a box we mean its sides that touch the ground and constitute its base area, and by height we mean the side orthogonal to the base area. Let

$$f : B \to \mathbb{R}_+$$

denote the optimal value function on B, i.e. it is the maximum height possible when the input is on top of a stack. Since $f(b) \geq h(b)$, we can initialize $f(b)$ to $h(b)$. What is the dynamic programming relationship? Let S denote an optimal stack where some $b \in B$ is on the top. After we remove b, the remaining stack $S - b$ should also be optimal. If it is not, then $S = (S - b) + b$ cannot be optimal as well. Note however that we have a constraint here for S: in order for b to be on the top, its width and depth must be strictly smaller than those of the box below. In summary, the dynamic programming relationship is (recall B is sorted)

$$f(b_j) = h(b_j) + \max\big\{f(b_i) \mid i < j,\ w(b_i) < w(b_j), d(b_i) < d(b_j)\big\}. \tag{4}$$

Thus, using Eq. (4) we can first calculate f for first few elements in B, then use them to calculate f for later elements in B. Finally, the optimal value is the maximum of f on B.

We see that the box stacking problem is very similar to the longest increasing subsequence problem (Section 5.2). In LIS we append larger elements one after another, and in box stacking problem we stack smaller boxes on top of others. For an optimal LIS s, the subsequence obtained by removing the last element in s should be optimal as well. In box stacking, removal of the top box of an optimal stack S should result in an optimal sub-stack. Both problems feature *constraints* as we have seen in Eq. (3) and Eq. (4). These constraints shrinks our search spaces, but they do not prevent us from applying dynamic programming techniques.

5.4 Longest Common Subsequence

Given two (finite) sequences x and y, we say that the sequence z is a *common subsequence* of x and y if it is a subsequence of x and a subsequence of y. In the longest common subsequence (LCS) problem, we are given two sequence of strings $x = (x_1, x_2, \dots, x_m)$ and $y = (y_1, y_2, \dots, y_n)$, and we wish to find the longest common subsequence of them. x and y may have different lengths. Again, let's think about small problems first. If either x or y is empty, then the algorithm should return 0. If $x = (x_1)$ and $y = (y_1)$ both have length 1, then the problem reduces to judging whether $x = y$. If $x = (x_1, x_2)$ and $y = (y_1)$, then the length of the LCS is either 0 or 1, by comparing y_1 to x_1 and x_2. Let $f[(x_1\,..\,x_i), (y_1\,..\,y_j)]$ denote the optimal value function for subsequences $(x_1\,..\,x_i)$ and $(y_1\,..\,y_j)$. We have the relation

$$f[(x_1\,..\,x_i), (y_1\,..\,y_j)] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ f[(x_1\,..\,x_{i-1}), (y_1\,..\,y_{j-1})] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\left\{ f[(x_1\,..\,x_i), (y_1\,..\,y_{j-1})], f[(x_1\,..\,x_{i-1}), (y_1\,..\,y_j)] \right\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Given this relation, we can then construct a table of size $m \times n$ and fill in the table from small i and j to larger one, until we have $f(m, n)$. As we can see in the code, the running time of the algorithm is $\Theta(mn)$.

```
LCS($X, Y$)
    $m, n = X.length, Y.length$
    Let $f[0\,..\,m, 0\,..\,n]$ be a new table
    for $i = 1$ to $m$ do
        $f[i, 0] = 0$
    for $j = 1$ to $n$ do
        $f[0, j] = 0$
    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            if $x_i == y_j$ then
                $f[i, j] = f[i-1, j-1] + 1$
            else if $f[i-1, j] > f[i, j-1]$ then
                $f[i, j] = f[i-1, j]$
            else$f[i, j] = f[i, j-1]$
    return $f[m, n]$
```

6 Markov Chains

We discuss Markov chains in this section.....we will mostly follow Wasserman 2004... we assume the state space is discrete, either $\mathcal{X} = \{1, \dots, N\}$ or $\mathcal{X} = \{1, 2, \dots, \}$, and the time is discrete.

6.1 Definitions

Definition 6.1. *A stochastic process* $\{X_n : n \in T\}$ *is called a* Markov chain *if*

$$\mathbb{P}\{X_n = x \mid X_0, \dots, X_{n-1}\} = \mathbb{P}\{X_n = x \mid X_{n-1}\}$$

for all n *and all* $x \in \mathcal{X}$. *We shall use* $\mathbb{P}$ *to denote the* transition matrix. *Its* (i, j) *element is*

$$p_{ij} = \mathbb{P}\{X_{n+1} = j \mid X_n = i\}.$$

We require $p_{ij} \geq 0$ *and* $\sum_j p_{ij} = 1$.

We let

$$p_{ij}^{(n)} = \mathbb{P}\{X_{m+n} = j \mid X_m = i\}$$

be the probability of going from state i to state j in n steps. Let $\mathbb{P}^{(n)}$ be the matrix whose (i, j) element is $p_{ij}^{(n)}$. Note $\mathbb{P}^{(1)} = \mathbb{P}$.

Theorem 6.2 (Chapman-Kolmogorov equation)**.** *The* n*-step probabilities satisfy*

$$p_{ij}^{(m+n)} = \sum_{k \in \mathcal{X}} p_{ik}^{(m)} \cdot p_{kj}^{(n)}.$$

Proof.

$$\begin{aligned} p_{ij}^{(m+n)} &= \mathbb{P}\{X_{m+n} = j \mid X_0 = i\} \\ &= \sum_{k \in \mathcal{X}} \mathbb{P}\{X_{m+n} = j, X_m = k \mid X_0 = i\} \\ &= \sum_{k \in \mathcal{X}} \mathbb{P}\{X_{m+n} = j \mid X_m = k, X_0 = i\}\mathbb{P}\{X_m = k \mid X_0 = i\} \\ &= \sum_{k \in \mathcal{X}} \mathbb{P}\{X_{m+n} = j \mid X_m = k\}\mathbb{P}\{X_m = k \mid X_0 = i\} \\ &= \sum_{k \in \mathcal{X}} p_{ik}^{(m)} p_{kj}^{(n)}. \end{aligned}$$

□

This is the equation for matrix multiplication. Take $m = 1, n = 1$ we get

$$\mathbb{P}^{(2)} = \mathbb{P}^{(1)} \cdot \mathbb{P}^{(1)} = \mathbb{P} \cdot \mathbb{P} = \mathbb{P}^2,$$

and so in general we have $\mathbb{P}^{(n)} = \mathbb{P}^n$.

Definition 6.3. *We say that* i reaches j *(or* j *is* accessible *from* i*) if* $p_{ij}^{(n)} > 0$ *for some* $n \geq 0$*, and we write* $i \to j$. *If* $i \to j$ *and* $j \to i$ *we write* $i \leftrightarrow j$ *and we say that* i *and* j communicate.

The relation "$i \sim j$ if $i \leftrightarrow j$" is an equivalence equation, so it partitions the state into disjoint classes. If all states communicate with each other, then we say the Markov chain is *irreducible*. A set of states is *closed* if, once you enter that set you never leave. A closed set consisting of a single state is called an *absorbing state*, or a *sink*. For example, for $\mathcal{X} = \{1, 2, 3, 4\}$ and

$$\mathbb{P} = \begin{pmatrix} \frac{1}{3} & \frac{2}{3} & 0 & 0 \\ \frac{2}{3} & \frac{1}{3} & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

the classes are $\{1, 2\}$, $\{3\}$ and $\{4\}$. State 4 is an absorbing state.

Definition 6.4. *State i is* recurrent *or* persistent *if*

$$\mathbb{P}\{X_n = i \textit{ for some } n \geq 1 \mid X_0 = i\} = 1.$$

Otherwise, it is transient.

If a state i is recurrent, then we will eventually return to that state. After the return, we can re-start the timer, so once again we will eventually return to i. So a recurrent state is a state that will be visited infinitely many times.

Theorem 6.5. *A state i is recurrent if and only if*

$$\sum_{n=0}^{\infty} p_{ii}^{(n)} = \infty.$$

It is transient if and only if

$$\sum_{n=0}^{\infty} p_{ii}^{(n)} < \infty.$$

Proof. We define

$$I_n = \begin{cases} 1 & \text{if } X_n = i \\ 0 & \text{if } X_n \neq i. \end{cases}$$

The number of times the chain is in state i is $\sum_{n=0}^{\infty} I_n$. If it is recurrent, then the sum is infinity, so

$$\mathbb{E}\left[\sum_{n=0}^{\infty} I_n \mid X_0 = i\right] = \sum_{n=0}^{\infty} \mathbb{E}\left[I_n \mid X_0 = i\right] = \sum_{n=0}^{\infty} \mathbb{P}\left\{I_n = 1 \mid X_0 = i\right\} = \sum_{n=0}^{\infty} p_{ii}^{(n)} = \infty.$$

□

For example, once can show that for this Markov chain

$$\mathbb{P} = \begin{pmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

all states are recurrent, and for

$$\mathbb{P} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & \frac{1}{2} \end{pmatrix}$$

there are two classes of recurrent states, $\{1, 2\}$, $\{3, 4\}$ and one class of transient state $\{5\}$. We list the following facts:

1. If state i is recurrent and $i \leftrightarrow j$, then j is recurrent.
2. If state i is transient and $i \leftrightarrow j$, then j is transient.
3. A finite Markov chain must have at least one recurrent state.
4. The states of a finite, irreducible Markov chain are all recurrent.
5. (Decomposition theorem) the state space $\mathcal{X}$ can be written as the disjoint union

$$\mathcal{X} = \mathcal{X}_T \bigcup \mathcal{X}_1 \bigcup \mathcal{X}_2 \bigcup \cdots$$

where $\mathcal{X}_T$ are the transient states and each $\mathcal{X}_i$ is a closed, irreducible set of recurrent states.

Example 6.6. *Let's consider a one dimensional random walk. The state space is $\mathcal{X} = \mathbb{Z}$ and the transition probabilities are*

$$\begin{cases} p_{i,i+1} := p = 1 - p_{i,i-1} & \\ p_{ij} = 0 & \text{otherwise.} \end{cases}$$

All states communicate, hence either all states are recurrent or all are transient. We shall show that only for $p = 1/2$ that all states are recurrent, and for all $p \neq 1/2$ all states are transient. To this end we compute $\sum_{n=0}^{\infty} p_{00}^{(n)}$. If we start at 0 and return back to 0, then we must went through some even number of steps: same steps going forward and backward. So this sum is equivalent to $\sum_{n=0}^{\infty} p_{00}^{(2n)}$. Now

$$p_{00}^{(2n)} = \binom{2n}{n} p^n (1-p)^n = \frac{(2n)!}{n!n!} p^n (1-p)^n$$

and use Stirling's formula

$$n! \sim n^{n+1/2} e^{-n} \sqrt{2\pi}$$

we have

$$p_{00}^{(2n)} \sim \frac{[4p(1-p)]^n}{\sqrt{\pi n}}.$$

If $p = 1/2$, then $4p(1-p) = 1$ so

$$\sum_{n=0}^{\infty} p_{00}^{(2n)} \sim \frac{1}{\sqrt{\pi n}} = \infty,$$

which means all states are recurrent according to Theorem 6.5. If $p \neq 1/2$, then $4p(1-p) < 1$, so the series converges, and consequently all states are transient.

Definition 6.7. *Suppose we start at $X_0 = i$. The* recurrence time *is*

$$T_{ii} = \min\{n > 0 : X_n = i\},$$

assuming X_n ever returns to i, and we define $T_{ii} = \infty$ otherwise. The mean recurrence time *of a recurrent state i is*

$$\mathbb{E}[T_{ii}] = \sum_{n=1}^{\infty} n f_{ii}(n)$$

where

$$f_{ii}(n) = \mathbb{P}\{X_1 \neq i, X_2 \neq i, \ldots, X_{n-1} \neq i, X_n = i \mid X_0 = i\}.$$

A recurrent state is null *if $\mathbb{E}[T_{ii}] = \infty$ and otherwise it is called* non-null *or* positive*.*

Lemma 6.8. *If a state is null and recurrent, then $p_{ii}^{(n)} \to \infty$.*

Lemma 6.9. *In a finite state Markov chain, all recurrent states are positive.*

Definition 6.10. *If $p_{ii}^{(n)} = 0$ whenever n is* not *divisible by d, and d is the largest integer with this property, then we call d the* period *of state i, i.e. $d = \gcd\{n : p_{ii}^{(n)} > 0\}$. State i is* periodic *if $d(i) > 1$ and* aperiodic *if $d(i) = 1$. A state is* ergodic *if it is positive recurrent and aperiodic. A Markov chain is ergodic if all its states are ergodic.*

Theorem 6.11. *For any irreducible ergodic Markov chain, the limiting distribution $\pi_j = \lim_{n\to\infty} p_{ij}^{(n)}, j \in \mathcal{X}$ exists and does not depend on i. It is the unique solution of $\mathbb{P}^T \Pi = \Pi$.*

Example 6.12 (Gambler's ruin problem)**.** *At each play, the gambler could win \$1 with probability p and lose with probability $1 - p$. Suppose a gambler starts with \$$i$. What is the probability that the gambler's fortune will reach value N before reaching 0? Let this probability be p_i. Then as $N \to \infty$,*

$$p_i = \begin{cases} 1 - \left(\dfrac{1-p}{p}\right)^i & \text{if } p > \dfrac{1}{2}; \\ 0 & \text{if } p \le \dfrac{1}{2}. \end{cases}$$

6.2 Markov chain Monte Carlo (MCMC)

Applications of MCMC:

- (Probability) Assume a finite state space Ω and a weight function $w : \Omega \to \mathbb{R}_+$. The goal is to design a sampling process which samples every element $x \in \Omega$ with probability $\frac{w(x)}{\sum_{x\in\Omega} w(x)}$. The problem is that $|\Omega| = 2^N$ could be very large, so it may not be feasible to compute the sum in the denominator.
- (Optimization) Let Ω be a set of feasible solutions to an optimization problem. The goal is $\max_{x\in\Omega} f(x)$. Using MCMC, we can sample from the distribution

$$\Lambda(x) = \frac{\lambda^{f(x)}}{Z}, \lambda > 1$$

  where $Z = \sum_{x\in\Omega} \lambda^{f(x)}$. If $\lambda = 1$, then the distribution Λ is uniform and there is no information at all. However, as we increase λ the distribution becomes concentrated around the optimal solutions.

Definition 6.13. *Let Π be a probability distribution over Ω. A Markov chain is said to be* reversible *with respect to Π if*

$$\forall x, y \in \Omega, \quad \Pi(x)p_{xy} = \Pi(y)p_{yx}.$$

Define $Q(x, y) := \Pi(x)p_{xy}$. If the Markov chain is reversible, then $Q(x, y) = Q(y, x)$. Thus we can represent a reversible Markov chain with an undirected graph with weight $Q(x, y)$. Also note that

$$p_{xy} = \frac{Q(x, y)}{\sum\limits_z Q(x, z)}.$$

$Q(x, y)$ is called the *ergodic flow*. It is the amount of probability mass flowing from x to y given $\Pi(x)$. The equation

$$\Pi(x)p_{xy} = \Pi(y)p_{yx}$$

is called *detailed balance*. We ca use the equation to design p_{xy}.

Gibbs distribution:

$$p_G(x) = \frac{1}{Z}e^{-\beta E(x)}, \quad \beta = \frac{1}{k_B T}.$$

- High temperature corresponds to $\beta \to 0$, and p_G converges to uniform distribution.
- Low temperature corresponds to $\beta \to \infty$, and p_G concentrates on the minimum of $E(x)$.

We want $\Pi_i = p_G(x_i) = \frac{1}{Z} e^{-\beta E(x_i)}$, where $\Pi = (\Pi_i)$ is the stationary distribution of the Markov chain. How should we choose the transition probabilities p_{ij}? Denote by $N_i = N(x_i)$ the expected number of times in which the state x_i is observed. We want

$$N_i \propto p_G(x_i).$$

Consider one update of the Markov chain:

$$N_i^{(t+1)} = N_i^{(t)} + \sum_{j \neq i} \left[N_j^{(t)} p_{ji} - N_i^{(t)} p_{ij} \right].$$

The stationary conditions are

$$\begin{cases} N_i^{(t+1)} = N_i^{(t)} \\ N_i \propto p_G(x_i). \end{cases}$$

We should have *global balance*

$$\sum_{j \neq i} \left[N_j^{(t)} p_{ji} - N_i^{(t)} p_{ij} \right] = 0.$$

However, this is too complicated. Instead we impose that each term in the summation is zero (*detailed balance*):

$$N_j^{(t)} p_{ji} = N_i^{(t)} p_{ij} \quad \forall i, j.$$

We want

$$p_G(x_j) \cdot p_{ji} = p_G(x_i) \cdot p_{ij} \quad \forall i, j \quad \Rightarrow \quad \frac{p_{ij}}{p_{ji}} = e^{-\beta [E(x_j) - E(x_i)]} = e^{-\beta \Delta E_{ji}}$$

Regardless of whether the state is continuous or discrete, all Markov chain methods consist of repeatedly applying stochastic updates until eventually the state begins to yield samples from the equilibrium distribution. Running the Markov chain until it reaches its equilibrium distribution is called burning in the Markov chain. After the chain has reached equilibrium, a sequence of infinitely many samples may be drawn from the equilibrium distribution. They are identically distributed, but any two successive samples will be highly correlated with each other. A finite sequence of samples may thus not be very representative of the equilibrium distribution. One way to mitigate this problem is to return only every n successive samples, so that our estimate of the statistics of the equilibrium distribution is not as biased by the correlation between an MCMC sample and the next several samples. Markov chains are thus expensive to use because of the time required to burn in to the equilibrium distribution and the time required to transition from one sample to another reasonably decorrelated sample after reaching equilibrium. If one desires truly independent samples, one can run multiple Markov chains in parallel. This approach uses extra parallel computation to eliminate latency. The strategy of using only a single Markov chain to generate all samples and the strategy of using one Markov chain for each desired sample are two extremes; deep learning practitioners usually use a number of chains that is similar to the number of examples in a minibatch and then draw as many samples as are needed from this fixed set of Markov chains. A commonly used number of Markov chains is 100.

Another difficulty is that we do not know in advance how many steps the Markov chain must run before reaching its equilibrium distribution. This length of time is called the mixing time . Testing whether a Markov chain has reached equilibrium is also difficult. We do not have a precise enough theory for guiding us in answering this question. Theory tells us that the chain will converge, but not much more. If we analyze the Markov chain from the point of view of a matrix A acting on a vector of probabilities v, then we know that the chain mixes when A t has effectively lost all the eigenvalues from A besides the unique eigenvalue of 1. This means that the magnitude of the second-largest eigenvalue will determine the mixing time. In practice, though, we cannot actually represent our Markov chain in terms of a matrix. The number of states that our probabilistic model can visit is exponentially large in the number of variables, so it is infeasible to represent v, A, or the eigenvalues of A. Because of these and other obstacles, we usually do not know whether a Markov chain has mixed. Instead, we simply run the Markov chain for an amount of time that we roughly estimate to be sufficient, and use heuristic methods to determine whether the chain has mixed. These heuristic methods include manually inspecting samples or measuring correlations between successive samples.[1]

[1] From Goodfellow, Bengio, and Courville 2016.

7 Selected Topics

7.1 Introduction to Randomized Algorithms

Here we discuss some examples of randomized algorithms. We should largely follow Cormen et al. 2009 and Motwani and Raghavan 1995. We distinguish between two types of randomized algorithms:

1. *Las Vegas algorithm.* It always produces the correct answer, but the running time is random.
2. *Monte Carlo algorithm.* The running time is bounded, but it has a chance of producing incorrect results.

Las Vegas algorithm can be converted to Monte Carlo algorithm by forcing it to output some random answer if a time bound is reached. Conversely, if a Monte Carlo algorithm produces correct answer with positive probability, and an efficient verification procedure exists for checking the answer, then we can run the Monte Carlo algorithm many times until the correct answer is obtained.

7.1.1 Randomized Quicksort

It is easy to think up an alternative scheme for quicksort discussed in Section 3.3: *instead of always picking the first element or the last element as the pivot, why can't we choose a random element as the pivot each time*? This gives rise to the randomized quicksort.

Algorithm 7.1.1 Randomized Quicksort

```
RANDQ($A$, $p$, $r$)
    if $q < r$ then
        $q$ = RANDP($A, p, r$)
        RANDQ($A, p, q - 1$)
        RANDQ($A, q + 1, r$)
RANDP($A$, $p$, $r$)
    $i$ = RANDOM($p, r$)
    exchange $A[r]$ with $A[i]$
    return PARTITION($A$, $p$, $r$)
```

It is an example of the *Las Vegas* algorithm, since it will eventually sort the input array just as quicksort does, but its running time in terms of number of comparisons may be random. Let X be the total number of comparisons. We shall derive that $\mathbb{E}[X] = O(n \lg n)$. In our analysis We assume each element in the array is distinct. Let $A = \{z_1, z_2, \ldots, z_n\}$ denote the *sorted* array and let $Z_{ij} = \{z_i, \ldots, z_j\}$.

Note that any z_i and z_j are compared at most once. This is because every element is only compared to some pivot element, and after the call the pivot stays there and is never compared to any other elements in the array. Thus, we can let $X_{ij} = \mathbb{1}[z_i \text{ is compared to } z_j]$ and then

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Taking expectations we get

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{P}\{z_i \text{ is compared to } z_j\}.$$

So what is the probability of z_i being compared to z_j? We recall that only pivots have the chance of being compared to other elements in the array, so if z_i is compared to z_j, then at least one of them must be pivot. On the other hand, if at some stage some $z_i < x < z_j$ in Z_{ij} is chosen as the pivot, then z_i and z_j would be put into different partitions and they will never be compared afterwards.

Prior to the point at which an element from Z_{ij} is chosen as a pivot, the whole set Z_{ij} is in the same partition. Therefore any element of Z_{ij} is equally likely to be the first one chosen as a pivot. The set Z_{ij} has $j - i + 1$ elements, so each has probability of $1/(j - i + 1)$ of being chosen as the pivot. We then have

$$\begin{aligned}\mathbb{P}\{z_i \text{ is compared to } z_j\} &= \mathbb{P}\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}\\ &= \mathbb{P}\{z_i \text{ is the first pivot chosen from } Z_{ij}\}\\ &\quad + \mathbb{P}\{z_j \text{ is the first pivot chosen from } Z_{ij}\}\\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1}\\ &= \frac{2}{j-i+1}.\end{aligned}$$

Now

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}\\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \quad (\text{let } k = j - i)\\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}\\ &= \sum_{i=1}^{n-1} O(\lg n)\\ &= O(n \lg n).\end{aligned}$$

7.1.2 k-th Order Statistics

Recall how we obtain the maximum and minimum of an array:

```
MINIMUM($A$)
    $min = A[1]$
    for $i = 2$ to $len(A)$ do
        if $min > A[i]$ then
            $min = A[i]$
    return $min$
```

and

```
MAXIMUM($A$)
    $max = A[1]$
    for $i = 2$ to $len(A)$ do
        if $max < A[i]$ then
            $max = A[i]$
    return $max$
```

The two procedures take $O(n)$ times, since we really have to go through each element of the array to determine minimum or maximum.

Now, how do we find the median of the array? Recall when the length of the array n is odd, the median is the $i = (n+1)/2$th smallest element, while if n is even then the medians are at $i = \lfloor (n+1)/2 \rfloor$ and $i = \lceil (n+1)/2 \rceil$ position in the sorted array. We could first sort the array and then take the middle element, but this would take $O(n \lg n)$ time. We now discuss a randomized algorithm that can find the k-th order statistics in expected $O(n)$ time.

The idea is the same as randomized quicksort: we choose one element from the array at random as the pivot and put smaller elements on the left, larger elements on the right, by comparing each $x \in A$ with the pivot:

1. If the pivot is at the k-th position then it is the element we want to find.
2. Else if k is smaller than the size of the left array, then k must lies in the left sub-array and so we recursively call our procedure on the left sub-array.
3. Else if k is larger than the size of the left array, then k must lies in the right sub-array and so we recursively call our procedure on the right sub-array.

Algorithm 7.1.2 Randomized k-th Order Statistics

```
ORDERSTAT($A, p, r, k$)
    if $p == r$ then
        return $A[p]$
    $q = $ RANDP($A, p, r$)          ☙ randomly choose an element.
    $left\text{-}size = q - p + 1$     ☙ size of the left sub-array.
    if $k == left\text{-}size$ then
        return $A[q]$          ☙ 1. found the $k$-th order statistics.
    else if $k < left\text{-}size$ then          ☙ 2. $k$-th order statistics on the left.
        return ORDERSTAT($A, p, q - 1, k$)
    else          ☙ 3. $k$-th order statistics on the right.
        return ORDERSTAT($A, q + 1, r, (k - left\text{-}size)$)
```

In the following analysis, we assume all elements in the array are distinct. We let $T(n)$ denote the running time of the algorithm. We shall prove that $\mathbb{E}[T(n)] = O(n)$. Since we pick an element randomly, the size of the left sub-array can be $1, 2, \ldots, n$ with equal probability $1/n$. We let

$$X_k = \mathbb{1}\{\text{the left-subarray has exactly } k \text{ elements}\}, \qquad k = 1, 2, \ldots, n$$

so that $\mathbb{E}[X_k] = 1/n$. We have

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot \{T[\max(k-1, n-k)] + O(n)\} \\ &= \sum_{k=1}^{n} X_k \cdot T[\max(k-1, n-k)] + O(n). \end{aligned}$$

Taking expectations, we get

$$\begin{aligned}
\mathbb{E}[T(n)] &\le \mathbb{E}\left[\sum_{k=1}^{n} X_k \cdot \{T[\max(k-1,n-k)] + O(n)\}\right] \\
&= \sum_{k=1}^{n} \mathbb{E}[X_k \cdot T[\max(k-1,n-k)]] + O(n) \\
&= \sum_{k=1}^{n} \mathbb{E}[X_k] \cdot \mathbb{E}[T[\max(k-1,n-k)]] + O(n) \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathbb{E}[T[\max(k-1,n-k)]] + O(n) \\
&\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathbb{E}[T(k)] + O(n).
\end{aligned}$$

We used the fact that X_k and $T[\max(k-1,n-k)]$ are independent.

Assume $\mathbb{E}[T(n)] \le cn$ for some constant c. Then

$$\begin{aligned}
\mathbb{E}[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\
&\le \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\
&= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) \\
&= c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) \\
&\le \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right).
\end{aligned}$$

We want for sufficiently large n the last expression is at most cn, i.e. $cn/4 - c/2 - an \ge 0$. Re-arrange the terms

$$n \ge \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

So if we assume $T(n) = O(1)$ for $n < 2c/(c-4a)$, then $\mathbb{E}[T(n)] = O(n)$.

7.1.3 Universal Hashing

Any fixed hash function may be vulnerable to adversary attacks. The attacker can choose inputs that are all hashed to the same slot, yielding an average retrieval time of $\Theta(n)$. To address this problem, we can periodically switch to a new random chosen hash function (we move all old addresses to new ones). To further avoid confusion, we quote a post from https://math.stackexchange.com/questions/103107/universal-hashing:

Yes, the hash table handler must guarantee that it goes to the same address when it tries to find data – the same address that that hash table handler used when it stored that data in the hash table. So it must use the same hash function both times.

If hypothetically the hash table handler were to roll the dice and pick a fresh new random hash function for every new data item that came in ... well, as you pointed out, that would cause problems.

Each hash table is associated with one and only one hash function. Every key used to store data in that hash table was hashed using that one hash function. Typically the hash table handler uses one chosen hash function for millions of keys.

Many hash table handlers periodically switch from one hash function to some other new hash function. Many hash table handlers – whether they use universal hashing or not – switch every time the number of data items stored in the hash table doubles. In addition, as Tim Duff pointed out, hash table handlers that use that use universal hashing also switch when they observe too many collisions. When a hash table handler that uses universal hashing decides it is time to switch (for either reason), the hash table handler rolls the dice and picks a new random hash function.

Every time the hash function changes, the hash table handler moves each and every data item out of the old address (the address indicated by the old hash function) and into the new address (the address indicated by the new hash function). Once all that shuffling around is complete, every data item stored in the hash table is in an address entirely determined by its key and the current latest hash function. So then the hash table handler can forget the old hash function, and do all new lookups using the new hash function.

Definition 7.1. *Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into $\{0, 1, \dots, m-1\}$. Such a collection is said to be* **universal** *if for each pair of keys $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$. In other words, if we randomly select an h from $\mathcal{H}$, then the probability of collision for this particular key pair (k, l) is at most*

$$\frac{\textit{number of } h \textit{ such that } h(k) = h(l)}{\textit{total number of } h\textit{'s}} = \frac{|\mathcal{H}|/m}{|\mathcal{H}|} = \frac{1}{m}$$

Equivalently,

$$\forall k, l \in U, k \neq l : \mathbb{P}(\textit{collision}) = \mathbb{P}_{h \in \mathcal{H}}(h(k) = h(l)) \leq \frac{1}{m}.$$

Suppose keys are all in the range 0 to $p - 1$ inclusive for some large prime p. A class of hash functions that is universal is

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\},$$

where

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m) \tag{5}$$

and

$$\mathbb{Z}_p = \{0, 1, \dots, p-1\}, \quad \mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}.$$

Here is a proof of the claim. For any two distinct key pairs k and l from $\mathbb{Z}_p$, $r = (ak + b) \bmod p \neq s = (al + b) \bmod p$, because $r - s \equiv a(k - l) \bmod p$ is not zero. The function

$$f : \mathbb{Z}_p \to \mathbb{Z}_p, f(k) = (ak + b) \bmod p \tag{6}$$

is thus injective, so it is also bijective. Thus the probability that k and l collide is equal to the probability that $r \equiv s \bmod m$.

In $\mathbb{Z}_p$, the maximum size of congruence class of $\mathbb{Z}_m$ is at most $\lceil p/m \rceil$ (an example would be $\mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$ and $m = 3$. Seven divided by three is two plus a remainder of one: $7 = 2 \times 3 + 1$, and we see that $\{0, 3, 6\}$ has length $\lceil 7/3 \rceil = 3$) Thus, for a given value of r, of the remaining $p - 1$ possible values of $s \in \mathbb{Z}_p \setminus \{r\}$, the number of values s such that $s \neq r$ and $s \equiv r \bmod m$ is at most

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p + m - 1)/m) - 1 \\ &= (p - 1)/m. \end{aligned}$$

The inequality follows from the following lemma.

Lemma 7.2. *Prove the following inequalities:*

$$\left\lceil \frac{a}{b} \right\rceil \leq \frac{a + (b - 1)}{b}$$

Proof. Let $q = \lceil \frac{a}{b} \rceil$. We have

$$\begin{gathered} q < \frac{a}{b} + 1 \\ \Downarrow \\ qb < a + b \\ \Downarrow \\ (q - 1)b < a \\ \Downarrow \\ (q - 1)b + 1 \leq a \\ \Downarrow \\ qb \leq a + b - 1 \\ \Downarrow \\ q \leq \frac{a + b - 1}{b} \end{gathered}$$

□

We divide the right side by the size of $\mathbb{Z}_p \setminus \{r\}$, which is $(p - 1)$, to get the (upper bound) probability that $s \equiv r \bmod m$:

$$\frac{\text{upper bound of } |\{s : s \equiv r \bmod m\}|}{|\mathbb{Z}_p \setminus \{r\}|} = \frac{(p - 1)/m}{(p - 1)} = \frac{1}{m}.$$

Thus we have proved that for any pair of distinct values $k, l \in \mathbb{Z}_p$,

$$\mathbb{P}\{h_{ab}(k) = h_{ab}(l)\} \leq \frac{1}{m},$$

so that $\mathcal{H}_{pm}$ is indeed universal.

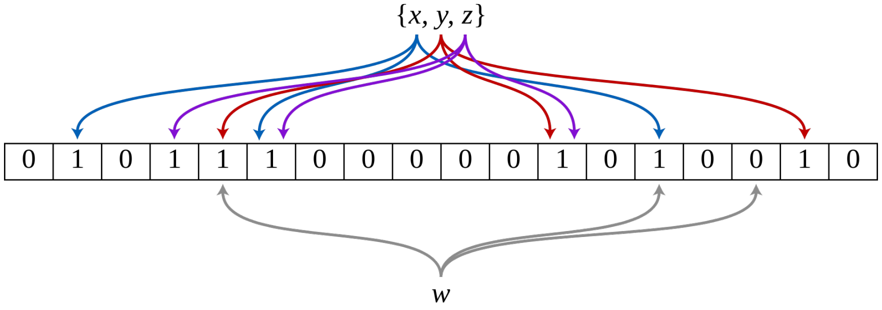


Figure 11: An example of a Bloom filter, representing the set x, y, z. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set x, y, z, because it hashes to one bit-array position containing 0. For this figure, $m = 18$ and $k = 3$.

7.1.4 Bloom Filters[2]

We often want to test whether a given element is in a set. For example, to test whether a word has correct spelling, we need to find if the word is in an English dictionary; in web scraping we need to test whether we have visited an URL address etc. We could use hash table, but when the number of elements in the set becomes very large, the storage requirement would also be huge. For email providers like Yahoo and Gmail, it has to test whether an email is a spam email. One way to to that is to store all the spam email addresses in a hash table, and test whether a given email address is in th hash table. But the hash table can grow to a size like hundreds of Gigabytes.

As another application, remember than when you register for a new account on some website, you may have the experience of choosing a username. When you type-in a username the website may tell you that it is already taken by somebody else. How can the website quickly determine whether a username is already taken? If we use linear search, we have to compare the typed username to millions of names in the database one-by-one, which takes a lot of time. If we use binary search tree to organize the usernames, then the search time is at best $O(\lg n)$. However, with Bloom filter, we are able determine whether a username is already taken in constant time! The price we may though, is that there is a small probability of being wrong, namely falsely report that the username is already taken while in fact it is not in the database.

Description of the Data Structure An empty Bloom filter is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions, generating a uniform random distribution. Typically, k is a constant, much smaller than m, which is proportional to the number of elements to be added; the precise choice of k and the constant of proportionality of m are determined by the intended false positive rate of the filter.

To *add* an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.

To *query* for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions. If any of the bits at these positions is 0, the element is definitely not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. In a simple Bloom filter, there is no way to distinguish between the two cases, but more advanced techniques can address this problem.

Removing an element from this simple Bloom filter is impossible because false negatives are not permitted. An element maps to k bits, and although setting any one of those k bits to zero suffices to remove the element, it also results in removing any other elements that happen to map onto that bit. Since there is no way of determining whether

[2] This part is taken from the Internet.

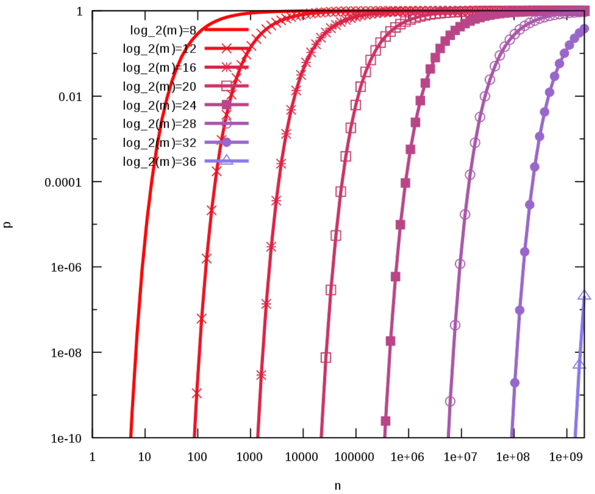


Figure 12: The false positive probability p as a function of number of elements n in the filter and the filter size m. An optimal number of hash functions $k = (m/n)\ln 2$ has been assumed.

any other elements have been added that affect the bits for an element to be removed, clearing any of the bits would introduce the possibility for false negatives.

Space and Time Advantages While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). However, Bloom filters do not store the data items at all, and a separate solution must be provided for the actual storage. Linked structures incur an additional linear space overhead for pointers. A Bloom filter with 1% error and an optimal value of k, in contrast, requires only about 9.6 bits per element, regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. The 1% false-positive rate can be reduced by a factor of ten by adding only about 4.8 bits per element.

However, if the number of potential values is small and many of them can be in the set, the Bloom filter is easily surpassed by the deterministic bit array, which requires only one bit for each potential element. Note also that hash tables gain a space and time advantage if they begin ignoring collisions and store only whether each bucket contains an entry; in this case, they have effectively become Bloom filters with $k = 1$.

Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant, $O(k)$, completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, however, the Bloom filter shines because its k lookups are independent and can be parallelized.

To understand its space efficiency, it is instructive to compare the general Bloom filter with its special case when k$=$ 1. If $k = 1$, then in order to keep the false positive rate sufficiently low, a small fraction of bits should be set, which means the array must be very large and contain long runs of zeros. The information content of the array relative to its size is low. The generalized Bloom filter (k greater than 1) allows many more bits to be set while still maintaining a low false positive rate; if the parameters (k and m) are chosen well, about half of the bits will be set, and these will be apparently random, minimizing redundancy and maximizing information content.

Probability of False Positive Assume a hash function selects each array position with equal probability. If m is the number of bits in the array, the probability that a certain bit is not set to 1 by a certain hash function during the insertion of an element is

$$1 - \frac{1}{m}.$$

If k is the number of hash functions and each has no significant correlation between each other, then the probability that the bit is not set to 1 by any of the hash functions is

$$\left(1 - \frac{1}{m}\right)^{k}.$$

If we have inserted n elements, the probability that a certain bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn}.$$

The probability that it is 1 is therefore

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$

Now test membership of an element that is not in the set. Each of the k array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is given as

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k}. \tag{7}$$

Be careful that $p(k) = f(g(k), k)$ where $g(k) = 1 - e^{-kn/m}$. The derivative of p with respect to k is

$$\frac{dp}{dk} = \frac{\partial f}{\partial g}\frac{dg}{dk} + \frac{\partial f}{\partial k} = \left(1 - \mathrm{e}^{-\frac{n}{m}k}\right)^{k}\left(\ln\left(1 - \mathrm{e}^{-\frac{n}{m}k}\right) + \frac{\frac{n}{m}k\mathrm{e}^{-\frac{n}{m}k}}{\left(1 - \mathrm{e}^{-\frac{n}{m}k}\right)}\right).$$

We want to find k so as to minimize the false positive probability $p(k)$, so we would like to find k such that $p'(k) = 0$. The expression in the right parenthesis must be zero. Since each k in the expression has n/m in front of it, we just need to find the solution k^* for

$$\ln\left(1 - \mathrm{e}^{-k}\right) + \frac{k\mathrm{e}^{-k}}{\left(1 - \mathrm{e}^{-k}\right)} = \ln\left(1 - \frac{1}{e^{k}}\right) + \frac{k}{e^{k} - 1} = 0 \tag{8}$$

and then set $\frac{m}{n}k^*$ for the original problem. The solution to Eq. (8) is guessed to be $\ln 2$. Hence, the optimal number of hash functions k that minimizes the false positive probability is

$$k^* = \frac{m}{n}\ln 2. \tag{9}$$

Substitute Eq. (9) into Eq. (7), we get

$$p^* = \left(1 - e^{-\ln 2}\right)^{\frac{m}{n}\ln 2} = 2^{-\frac{m}{n}\ln 2} \Rightarrow \ln p = -\frac{m}{n}(\ln 2)^2.$$

From above, given the target false positive probability p, we can get the optimal number of bits per element:

$$\frac{m}{n} = -\frac{\ln p}{(\ln 2)^2} \approx -1.44\log_2 p$$

and the optimal number of hash functions as

$$k = -\frac{\ln p}{\ln 2} = -\log_2 p.$$

7.1.5 Karger's min-cut Algorithm[3]

Let $G = (V, E)$ be a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges $C \subset E$ whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimum cardinality. We now study a simple algorithm for finding a min-cut of a graph.

We repeat the following step: pick an edge uniformly at random and merge the two vertices at its end-points (). If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Edges between vertices that are merged are removed, so that there are never any self-loops. We refer to this process of merging the two end-points of an edge into a single vertex as the *contraction* of that edge. With each contraction, the number of vertices of G decreases by one. The crucial observation is that an edge contraction does not reduce the min-cut size in G. This is because every cut in the graph at any intermediate stage is a cut in the original graph. The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut.

```
CONTRACT($G$)
  while $|V| > 2$ do
    Choose $e \in E$ uniformly at random
    $G = G/e$
  return the only cut $C$ in $G$
```

Let $n = |V|$ and let $k = |C|$ be the min-cut size. We fix our attention on a particular min-cut C with k edges. Notice that each vertex must have at least k edges, otherwise there would be a min-cut of size less than k, so that there are at least $kn/2$ edges in the graph.

Let $\mathcal{E}_i$ denote the event of *not* picking an edge of C at the ith step, for $1 \leq i \leq n-2$. The probability that the edge randomly chosen in the first step is in C is at most $k/(nk/2) = 2/n$, so that

$$\mathbb{P}[\mathcal{E}_1] \geq 1 - \frac{2}{n}.$$

Assuming that $\mathcal{E}_i$ occurs, during the second step there are at least $k(n-1)/2$ edges, so the probability of picking an edge in C is at most $2/(n-1)$, so that

$$\mathbb{P}[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - \frac{2}{n-1}.$$

At the ith step, the number of remaining vertices is $n - i + 1$. The size of the min-cut is still at least k, so the graph has at least $k(n-i+1)/2$ edges remaining at this step. Thus]

$$\mathbb{P}\left[\mathcal{E}_2 \mid \bigcap_{j=1}^{i-1} \mathcal{E}_j\right] \geq 1 - \frac{2}{n-i+1}.$$

The probability that no edge of C is ever picked in the process is

$$\mathbb{P}\left[\bigcap_{i=1}^{n-2} \mathcal{E}_i\right] = \prod_{i=1}^{n-2} \mathbb{P}\left[\mathcal{E}_i \mid \bigcap_{j=1}^{i-1} \mathcal{E}_j\right] \geq \prod_{i=1}^{2}\left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}.$$

The probability that the output is a min-cut is larger than $2/n^2$. Thus we can repeat the algorithm $n^2/2$ times, making independent random choices each time. The probability that a min-cut is not found in any of the $n^2/2$ attempts is at most

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} < \frac{1}{e}.$$

Further executions of the algorithm will make the failure probability arbitrarily small.

[3] This part is taken from Motwani and Raghavan 1995.

7.2 The PageRank algorithm

We now discuss the PageRank algorithm used by Google (Brin and Page 1998) to rank web pages returned by keyword queries. Our notation will be different from that of the original paper (Page et al. 1999).

Let $G = (V, E)$ denote the directed graph of web pages on the Internet, where $V = \{0, 1, \dots, n\}$, and $(i, j) \in E$ if in page i there is a link to page j. The goal of the PageRank algorithm is to compute a function

$$r : V \to \mathbb{R}_+$$

on V such that $r(i)$ is large if

1. either many web pages link to i (i.e. the in-degree of i is large), or
2. among links to i there are pages with high in-degrees.

To achieve these desired properties, we use a simple Markov chain model. Let p_{ij} denote the probability of jumping from i to j, so that $\sum_{j \in V} p_{ij} = 1$. We use simple heuristics to set p_{ij}:

- Page $0 \in V$ is the Google homepage and we assume it is linked from and to all pages in V. A surfer, no matter where he is, jumps to 0 when he is browsing the web with probability $1 - c$. Here c is a constant (we let it stand for "continue"). Thus
$$p_{i0} = 1 - c \quad \text{for any } i \in V.$$
- For any page $i \in V$, the probabilities of jumping from i to all the links in i are equal. In notation
$$p_{ij} = \frac{c}{n_i} \mathbb{1}[i \to j], \quad j \in V \setminus \{0\} \tag{10}$$
where n_i is the number of links in i (out-degrees). In this way
$$\sum_{j \in V} p_{ij} = p_{i0} + \sum_{j \in V \setminus \{0\}} p_{ij} = (1 - c) + \frac{c}{n_i} \cdot n_i = (1 - c) + c = 1.$$
In particular, for page 0 we have $p_{0j} = c/n$ for all $j = 1, \dots, n$.

Let $\mathbb{P}$ denote the transition matrix (p_{ij}), and let $\Pi = (\Pi_i)_{j \in V}$ denote the stationary distribution on V. We have $\Pi = \mathbb{P}^T \Pi$, i.e. Π is an eigenvector of $\mathbb{P}^T$. The stationary distribution Π can be obtained by first initializing the vector randomly to some $\Pi^{(0)}$ and then repeatedly applying the matrix $\mathbb{P}^T$, i.e. $\Pi = \lim_{n \to \infty} (\mathbb{P}^T)^n \Pi^{(0)}$. For $j \neq 0$ we have

$$\Pi_j = \Pi_0 \cdot p_{0j} + \Pi_1 \cdot p_{1j} + \dots + \Pi_n \cdot p_{nj} = (1 - c)\frac{c}{n} + \sum_{i \in V \setminus \{0\}} \Pi_i \cdot \frac{c}{n_i} \mathbb{1}[i \to j].$$

Algorithm 7.2.1 PageRank Algorithm

```
PAGERANK($G$)
    Randomly initialize some distribution $r : V \to \mathbb{R}_+$ on $V$
    Let $\mathbb{P} = (p_{ij})$ where $p_{ij}$ is as in Eq. (10)
    while not converge do
        $r = \mathbb{P}^T r$
    return the vector $r : V \to \mathbb{R}_+$
```

We can let $r(j) := \Pi_j \cdot \frac{c}{n}$ so that we get the classical equation

$$r(j) = (1 - c) + c \cdot \sum_{i \in V \setminus \{0\}} \frac{r(i)}{n_i} \mathbb{1}[i \to j].$$

We see that indeed $r(j)$ would be large if there are many nonzero terms in the sum, i.e. $\mathbb{1}[i \to j] \neq 0$ for many $i \in V \setminus \{0\}$, or some $r(i)$ is large in the summation.

7.3 Gradient Descent

Gradient descent is a popular numerical optimization technique for training neural networks. Here we present several variants of the gradient descent algorithm.

7.3.1 Batch Gradient Descent

Repeat until convergence:

◇ $\theta = \theta - \epsilon \cdot \frac{1}{n} \nabla_\theta \sum_{i=1}^{n} L(f(x_i; \theta), y_i)$

7.3.2 Stochastic Gradient Descent

Repeat until convergence:

1. Randomly sample one example i from the training set
2. $\theta = \theta - \epsilon \nabla_\theta L(f(x_i; \theta), y_i)$

7.3.3 Mini-batch gradient descent

Repeat until convergence:

1. Randomly sample a minibatch of m examples from the training set
2. $\theta = \theta - \epsilon \cdot \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x_i; \theta), y_i)$

7.3.4 Momentum

The momentum method uses physical heuristics to update the parameters. It views the cost $J(\theta)$ as energy, so that $-\nabla_\theta J(\theta)$ is analogous to force. Consider a discrete-time setting. Let θ_t, v_t, and a_t denote the position, velocity and acceleration at time t respectively. Set the mass of the particle to 1 so that acceleration is equal to the force in magnitude. The velocity at t is the velocity at time $t-1$ plus the acceleration at time $t-1$, and the position at time $t+1$ is the current position plus the velocity at time t:

$$v_t = v_{t-1} + a_{t-1}, \tag{11}$$

$$\theta_{t+1} = \theta_t + v_t. \tag{12}$$

According to the analogy above, the update rule of the momentum method is given by

```
Repeat until convergence:
    1. Randomly sample a minibatch of $m$ examples from the training set
    2. $v = \alpha v - \epsilon \cdot \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x_i; \theta), y_i),$
    3. $\theta = \theta + v$
```

Here $\alpha \in [0, 1)$ is a hyperparameter that determines how quickly the contributions of previous gradients exponentially decay.

Nesterov momentum:

```
Repeat until convergence:
    1. Randomly sample a minibatch of $m$ examples from the training set
    2. $v = \alpha v - \epsilon \cdot \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x_i; \theta + \alpha v), y_i),$
    3. $\theta = \theta + v$
```

7.3.5 AdaGrad

```
Repeat until convergence:
    1. Randomly sample a minibatch of $m$ examples from the training set
    2. $g = \epsilon \cdot \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x_i; \theta), y_i)$
    3. $r = r + g \odot g$
    4. $\theta = \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g$ (division and square root applied element-wise)
```

7.3.6 RMSProp

```
Repeat until convergence:
    1. Randomly sample a minibatch of $m$ examples from the training set
    2. $g = \cdot \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x_i; \theta), y_i)$
    3. $r = \rho r + (1 - \rho) g \odot g$
    4. $\theta = \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g$ (division and square root applied element-wise)
```

7.3.7 Adam

```
for $t = 0$ to $M$:
    1. Randomly sample a minibatch of $m$ examples from the training set
    2. $g = \frac{1}{m}\nabla_\theta \sum_{i=1}^{m} L(f(x_i;\theta), y_i)$
    3. $v = [\beta_1 v + (1-\beta_1)g]/(1-\beta_1^t)$
    4. $r = [\beta_2 r + (1-\beta_2)g \odot g]/(1-\beta_2^t)$
    5. $\theta = \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot v$ (operations applied element-wise)
```

Default settings of the 4 hyper-parameters proposed in Kingma and Ba 2014 are $\epsilon = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\delta = 10^{-8}$.

References

Brin, S. and L. Page (1998). “The Anatomy of a Large-scale Hypertextual Web Search Engine”. In: *Proceedings of the Seventh International Conference on World Wide Web 7*. WWW7. Brisbane, Australia: Elsevier Science Publishers B. V., pp. 107–117.

Cormen, T. H. et al. (2009). *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Kingma, D. P. and J. Ba (2014). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].

Motwani, R. and P. Raghavan (1995). *Randomized Algorithms*. New York, NY, USA: Cambridge University Press.

Page, L. et al. (Nov. 1999). *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab.

Wasserman, L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer Texts in Statistics. New York: Springer.