数据结构与算法 B Cheat Sheet

1 绪论

1.1 算法的时间复杂度及其表示法

什么是算法

算法是对计算过程的描述，是为了解决某个问题而设计的有限长操作序列。

算法的性质

有穷性：一个算法必须可以用有穷条指令描述，且必须在执行有穷次操作后终止。每次操作都必须在有穷时间内完成。算法终止后必须给出所处理问题的解或宣告问题无解。

确定性：一个算法，对于相同的输入，无论运行多少次，总是得到相同的输出。也可以说只要算法运行前的初始条件相同，那么算法运行的结果也相同。

可行性：算法中的指令（或描述语句）含义明确无歧义，且可以被机械化地自动执行。

输入/输出：输入指的是描述算法所处理的问题的数据，输出指的是描述该问题的答案的数据。算法可以不需要输入。但是没有输出的算法是没有意义的。

程序或算法的时间复杂度

一个程序或算法的时间效率，也称“时间复杂度”，有时简称“复杂度”。复杂度常用大的字母 O 和小写字母 n 来表示，比如 $O(n),O(n^2)$ 等。n 代表问题的规模，$O(X)$ 就表示解决问题的时间和 X 成正比关系（粗略理解）。

时间复杂度是用算法运行过程中，某种时间固定的操作需要被执行的次数和 n 的关系来度量的。在无序数列中查找某个数，复杂度是 $O(n)$。

计算复杂度的时候，只统计执行次数最多的 (n 足够大时) 那种固定操作 (称为基本操作) 的次数。比如某个算法需要执行加法 n^2 次，除法 $10000n$ 次，那么就记其复杂度是 $O(n^2)$ 的。

如果复杂度是多个 n 的函数之和，则只关心随 n 的增长增长得最快的那个函数。

程序或算法的时间复杂度

在无序数列中查找某个数 (顺序查找)：$O(n)$

插入排序、选择排序等笨排序方法：$O(n^2)$

快速排序：$O(n\log(n))$

二分查找：$O(\log(n))$

Python 中一些操作的时间复杂度总结

$O(1)$ 复杂度的常见操作：

1) 根据下标访问列表、字符串、元组中的元素
2) 在集合、字典中增删元素
3) 调用列表的 `append` 函数在列表末尾添加元素，以及用 `pop()` 函数删除列表末尾元素
4) 用 `in` 判断元素是否在集合中或某关键字是否在字典中
5) 以关键字为下标访问字典中的元素的值
6) 用 `len` 函数求列表、元组、集合、字典的元素个数

$O(n)$ 复杂度的常见操作：

1) 用 `in` 判断元素是否在字符串、元组、列表中
2) 用 `insert` 在列表中插入元素
3) 用 `remove` 或 `del` 删除列表中的元素
4) 用字符串、元组或列表的 `find`、`rfind`、`index`等函数做顺序查找
5) 用字符串、元组或列表的 `count` 函数计算元素出现次数
6) 用 `max`, `min`函数求列表、元组的最大值，最小值
7) 列表和元组加法

$O(n\log n)$ 复杂度的常见操作：

Python 自带排序 `sort`, `sorted`

$O(\log n)$ 复杂度的常见操作：

在排好序的列表或元组上进行二分查找（初始的查找区间是整个元组或列表，每次和查找区间中点比较大小，并缩小查找区间到原来的一半。类似于查英语词典）有序就会找得快！Pyhon 并不自带二分查找函数。

`in` 用于列表和用于字典、集合的区别

`a in b`

若 `b` 是列表，字符串或元组，则该操作时间复杂度 $O(n)$，即时间和 `b` 的元素个数成正比

若 `b` 是字典或集合，则该操作时间复杂度 $O(1)$，即时间基本就是常数，和 `b` 里元素个数无关

因此集合用于需要经常判断某个东西是不是在一堆东西里的情况

此种场合用列表替代集合，容易导致超时!!!!

最坏复杂度、平均复杂度、最好复杂度

算法的复杂度有最好情况下复杂度、最坏情况下的复杂度和平均复杂度之分，虽然许多情况下最坏复杂度和平均复杂度恰好相同。

快速排序为例，一般情况下待排序序列杂乱无章，这种情况下快速排序的复杂度就是平均复杂度 $O(n\log(n))$，但是在待排序的序列处于基本有序或基本逆序的最坏情况下，其复杂度会变成 $O(n^2)$。

1.2 数据的逻辑结构和存储结构

什么是数据结构？

数据结构 (data structure) 就是数据的组织和存储形式。描述一个数据结构，需要指出其逻辑结构、存储结构和可进行的操作。

将数据的单位称作“元素”或“结点”。数据结构描述的就是结点之间的关系。

数据的逻辑结构

从逻辑上描述结点之间的关系，和数据的存储方式无关。

集合结构：结点之间没有什么关系，只是属于同一集合。如 `set`。

线性结构：除了最靠前的结点，每个结点有唯一前驱结点；除了最靠后的结点，每个结点有唯一后继结点。如 `list`。

树结构：有且仅有一个结点称为”根结点”，其没有前驱 (父结点)；有若干个结点称为“叶结点”，没有后继 (子结点)；其它结点有唯一前驱，有 1 个或多个后继。如家谱。

图结构：每个结点都可以有任意多个前驱和后继，两个结点还可以互为前驱后继。如铁路网，车站是结点。

数据的存储结构

数据在物理存储器上存储的方式，大部分情况下指的是数据在内存中存储的方式。

顺序结构：结点在内存中连续存放，所有结点占据一片连续的内存空间。如 `list`。

链接结构：结点在内存中可不连续存放，每个结点中存有指针指向其前驱结点和/或后继结点。如链表，树。

索引结构：将结点的关键字信息（比如学生的学号）拿出来单独存储，并且为每个关键字 `x` 配一个指针指向关键字为 `x` 的结点，这样便于按照关键字查找到相应的结点。

散列结构：设置散列函数，散列函数以结点的关键字为参数，算出一个结点的存储位置。

数据的逻辑结构和存储结构无关

一种逻辑结构的数据，可以用不同的存储结构来存储。

树结构、图结构可以用链接结构存储，也可以用顺序结构存储。

线性结构可以用顺序结构存储，也可以用链接结构存储。

数据结构上的操作

建立（初始化）

插入结点

删除结点

查找结点

求结点前驱或结点后继。如线性表、树和图。

随机访问。即“找第 `i` 个结点”，如顺序表。

掌握一个数据结构，不但要了解其逻辑结构、存储结构，以及其上进行的各种操作，还需要知道每种操作的时间复杂度。

2 线性表

2.1 顺序表

即 Python 的列表，以及其它语言中的数组

元素在内存中连续存放

每个元素都有唯一序号（下标），且根据序号访问（包括读取和修改）元素的时间复杂度是 $O(1)$ 的 — 随机访问

下标为 `i` 的元素前驱下标为 `i-1`，后继下标为 `i+1`

顺序表支持的操作

序号	操作	含义	时间复杂度
1	init(n)	生成一个n个元素的顺序表，元素值随机	O(1)
2	init(a_0,a_1,....a_n)	生成元素为a_0,a_1,.... a_n的顺序表	O(n)
3	length()	求表中元素个数	O(1)
4	append(x)	在表的尾部添加一个元素x	O(1)
5	pop()	删除表尾元素	O(1)
6	get(i)	返回下标为i的元素	O(1)
7	set(i,x)	将下标为i的元素设置为x	O(1)
8	find(x)	查找元素x在表中的位置	O(n)
9	insert(i,x)	在下标i处插入元素x	O(n)
10	remove(i)	删除下标为i的元素	O(n)

顺序表的 `append` 的 $O(1)$ 复杂度的实现

总是分配多于实际元素个数的空间（容量大于元素个数）

元素个数小于容量时，`append` 操作复杂度 $O(1)$

元素个数等于容量时，`append` 导致重新分配空间，且要拷贝原有元素到新空间，复杂度 $O(n)$

重新分配空间时，新容量为旧容量的 k 倍 ($k>1$ 且固定)，可确保 `append` 操作的平均复杂度是 $O(1)$。Python 的 `list` 取 $k=1.2$ 左右。

2.2 链表

元素在内存中并非连续存放，元素之间通过指针链接起来

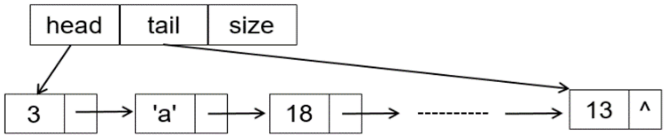
每个结点除了元素，还有 `next` 指针，指向后继

不支持随机访问。访问第 `i` 个元素，复杂度为 $O(n)$

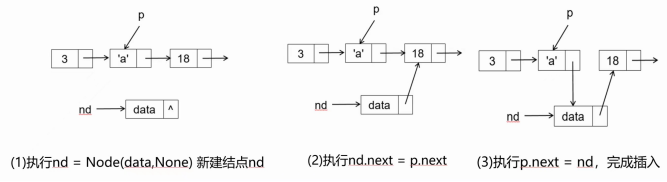
已经找到插入或删除位置的情况下，插入和删除元素的复杂度 $O(1)$, 且不需要复制或移动结点

有多种形式：单链表、循环单链表、双向链表、循环双向链表

单链表




```
class LinkList:
    class Node: #表结点
        def __init__(self, data, next=None):
            self.data, self.next = data, next
    def __init__(self):
        self.head = self.tail = None
        self.size = 0

    def printList(self): #打印全部结点
        ptr = self.head
        while ptr is not None:
            print(ptr.data, end=",")
            ptr = ptr.next

    def insert(self,p,data): #在结点p后面插入元素
        nd = LinkList.Node(data,None)
        if self.tail is p: # 新增的结点是新表尾
            self.tail = nd
        nd.next = p.next
        p.next = nd
        self.size += 1

    def delete(self,p): #删除p后面的结点
        if self.tail is p.next:
            self.tail = p
        p.next = p.next.next
        self.size -= 1

    #结点空间会被Python自动回收
```



(1)执行nd = Node(data,None) 新建结点nd　(2)执行nd.next = p.next　(3)执行p.next = nd，完成插入

(1)初始状态，将要删除'a'结点　(2)执行p.next = p.next.next，完成删除

判断变量是否为 None，应写 `p is None`，`p is not None` 最好不要写 `p == None`，`p != None`

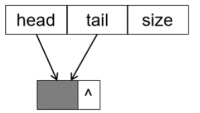
```
    def popFront(self): #删除前端元素
        if self.head is None:
            raise \
              Exception("Popping front for Empty link list.")
        else:
            self.head = self.head.next
            self.size -= 1
            if self.size == 0:
                self.tail = self.head = None
    def pushBack(self,data): #在尾部添加元素
        if self.size == 0:
            self.pushFront(data)
        else:
            self.insert(self.tail,data)

    def pushFront(self,data): #在链表前端插入一个元素DATA
        nd = LinkList.Node(data, self.head)
        self.head = nd
        self.size += 1
        if self.tail is None:
            self.tail = nd

    def clear(self):
        self.head = self.tail = None
        self.size = 0
    def __iter__(self):
        self.ptr = self.head
        return self
    def __next__(self):
        if self.ptr is None:
            raise StopIteration() # 引发异常
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data
```

```
linkLst = LinkList()
linkLst.pushFront(0)
linkLst.pushFront(1)
for i in range(2,5):
    linkLst.pushBack(i)
for x in linkLst: #>>1,0,2,3,4,
    print(x,end=",")
```

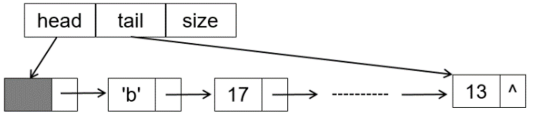
上述实现方式没有实现“隐藏”，不是很好的实现方式

带头结点的单链表

为避免链表为空是做特殊处理，可以为链表增加一个空闲头结点



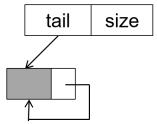
带头结点的空单链表



带头结点的非空单链表

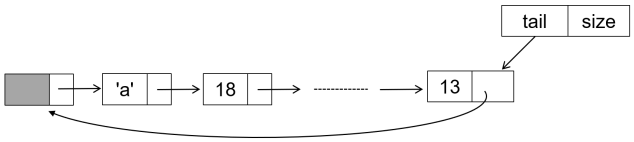
```
class LinkList:
    def __init__(self):
        self.head = self.tail = LinkList.Node(None,None)
        self.size = 0
```

循环单链表



空表

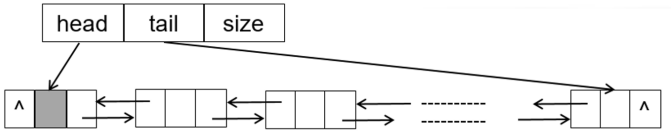
循环单链表在表首或表尾添加元素，以及删除表首元素，复杂度都是O(1)的。



tail.next即头结点

双向链表

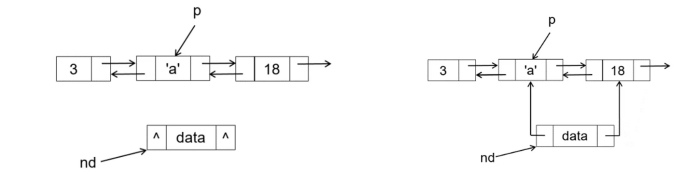
每个结点有 text 指针指向后继，有 prev 指针指向前驱



带头结点的双向链表

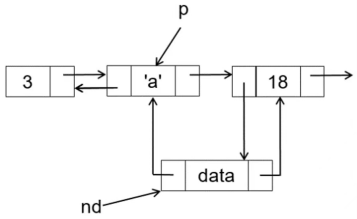
```
class DoubleLinkList:
    class _Node:
        def __init__(self, data, prev=None, next=None):
            self.data, self.prev, self.next = data, prev,
                next
```

在结点 p 后面插入新结点 nd

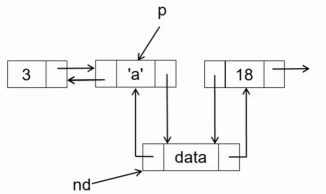


(1)执行nd = Node(data,None,None) 新建结点nd

(2) 执行 nd.prev, nd.next = p, p.next

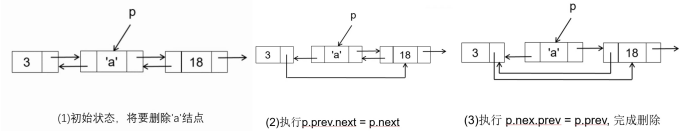


(3)执行p.next.prev = nd



(4)执行p.next = nd，插入完成

删除结点 p



双向链表实现

```
class DoubleLinkList:
    class _Node:
        def __init__(self, data, prev=None, next=None):
            self.data, self.prev, self.next = data, prev,
                 next

    class _Iterator:
        def __init__(self,p):
            self.ptr = p
        def getData(self):
            return self.ptr.data
        def setData(self,data):
            self.ptr.data = data
        def __next__(self):
            self.ptr = self.ptr.next
            if self.ptr is None:
                return None
            else:
                return DoubleLinkList._Iterator(self.ptr)
        def prev(self):
            self.ptr = self.ptr.prev
            return DoubleLinkList._Iterator(self.ptr)

    def __init__(self):
        self._head = self._tail = \
            DoubleLinkList._Node(None,None,None)
        self._size = 0

    def _insert(self,p,data):
        nd = DoubleLinkList._Node(data,p,p.next)
        if self._tail is p: # 新增的结点是新表尾
            self._tail = nd
        if p.next:
            p.next.prev = nd
        p.next = nd
        self._size += 1

    def _delete(self,p): #删除结点p
        if self._size == 0 or p is self._head:
            raise Exception("Illegal deleting.")
        else:
            p.prev.next = p.next
            if p.next: #如果p有后继
                p.next.prev = p.prev
            if self._tail is p:
                self._tail = p.prev
            self._size -= 1

    def clear(self):
        self._tail = self._head
        self._head.next = self._head.prev = None
        self.size = 0

    def begin(self):
        return DoubleLinkList._Iterator(self._head.next)

    def end(self):
        return None

    def insert(self,i,data): #在迭代器i指向的结点后面插入
         元素
        self._insert(i.ptr,data)

    def delete(self, i): # 删除迭代器i指向的结点
        self._delete(i.ptr)

    def pushFront(self,data): #在链表前端插入一个元素
        self._insert(self._head,data)

    def popFront(self):
        self._delete(self._head.next)

    def pushBack(self,data):
        self._insert(self._tail,data)

    def popBack(self):
        self._delete(self._tail)

    def __iter__(self):
        self.ptr = self._head.next
        return self

    def __next__(self):
        if self.ptr is None:
            raise StopIteration() # 引发异常
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data

    def find(self,val): #查找元素val，找到返回迭代器，找不
         到返回None
        ptr = self._head.next
        while ptr is not None:
            if ptr.data == val:
                return DoubleLinkList._Iterator(ptr)
            ptr = ptr.next
        return self.end()

    def printList(self):
        ptr = self._head.next
        while ptr is not None:
            print(ptr.data,end=",")
            ptr = ptr.next

linkLst = DoubleLinkList()
for i in range(5):
    linkLst.pushBack(i)
i = linkLst.begin()
while i != linkLst.end(): #>>0,1,2,3,4,
    print(i.getData(),end = ",")
    i = next(i)
print()
i = linkLst.find(3)
i.setData(300)
linkLst.printList() #>>0,1,2,300,4,
print()
linkLst.insert(i,6000) #在i后面插入6000
linkLst.printList() #>>0,1,2,300,6000,4,
print()
linkLst.delete(i)
linkLst.printList() #>>0,1,2,6000,4,
```

2.3 链表和顺序表的选择

顺序表

中间插入太慢

链表

访问第 i 个元素太慢

顺序访问也慢 (现代计算机有 cache，访问连续内存域比跳着访问内存区域快很多)

还多费空间

结论

尽量选用顺序表。比如栈和队列，都没必要用链表实现

基本只有在找到一个位置后反复要在该位置周围进行增删，才适合用链表

实际工作中几乎用不到链表

3 枚举与二分法

3.1 二分法寻找答案的核心思想

如果一个假设的答案成立，那就跳着试一个更优的假设答案看行不行；
如果一个假设的答案不成立，那就跳着试一个更差的假设答案看行不行。
必须每次验证假设答案，都可以把假设答案所在的区间缩小为上次的一半。
前提：单调性。一个假设答案不成立，则比它更优的假设答案肯定都不成立。

3.2 二分查找函数

写一个函数 BinarySeach，在从小到大排序的列表 a 里查找元素 p, 如果找到，则返回元素下标，如果找不到，则返回 None。要求复杂度 $O(\log(n))$

```
def binarySearch(a,p,key = lambda x : x):
    L, R = 0,len(a)-1 #查找区间的左右端点，区间含右端点
    while L <= R: #如果查找区间不为空就继续查找
        mid = L+(R-L)//2 #取查找区间正中元素的下标
        if key(p) < key(a[mid]):
            R = mid - 1 #设置新的查找区间的右端点
        elif key(a[mid]) < key(p):
            L = mid + 1 # 设置新的查找区间的左端点
        else:
            return mid
    return None
```

4 递归和分治

4.1 递归

递归的作用

1) 替代多重循环进行枚举
2) 解决本来就是用递归形式定义的问题
3) 将问题分解为规模更小的子问题进行求解
……

5 栈和队列

5.1 栈

类似于子弹匣，后压进去的子弹，先射出去
支持四种操作：

top()	返回栈顶元素
push(x)	将 x 压入栈中
pop()	弹出并返回栈顶元素
isEmpty()	看栈是否为空

要求上面操作复杂度都是 $O(1)$
用列表可以实现栈
四种操作的实现（stack 为一个列表）：

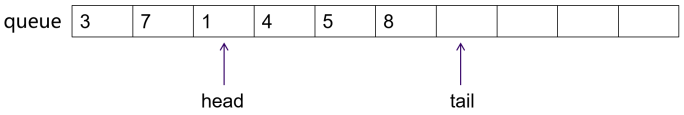
top()	stack[-1]
push(x)	stack.append(x)
pop()	stack.pop()
isEmpty()	len(stack) == 0

5.2 队列

即排队的队列。只能一头进 (push)，另一头出 (pop)。先进先出
要求进出的复杂度都是 $O(1)$
如果用列表的 append 进，pop(0) 出，则出的复杂度为 $O(n)$

队列的实现方法一

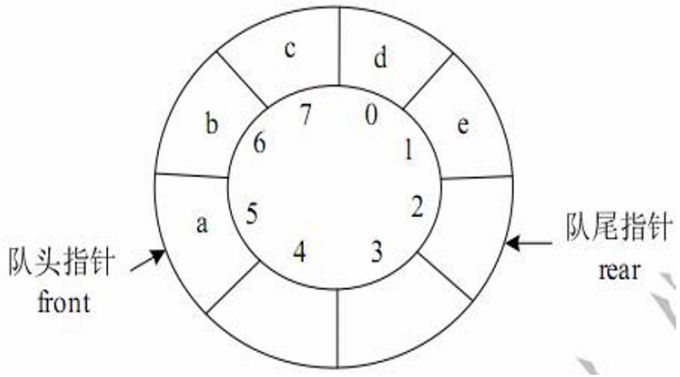
用足够大的列表实现，维护一个队头指针和队尾指针，初始：
head=tail = 0



head 指向队头元素，tail 指向队尾元素的后面
push(x) 的实现：
queue[tail] = x
tail+=1
pop() 的实现：head += 1
判断队列是否为空：head == tail

队列的实现方法二

如果不想浪费空间开足够大的列表，而是想根据实际情况分配空间，则可以用列表 + 头尾循环法实现队列



1) 预先开设一个 capacity 个空元素的列表 queue，head = tail = 0
2) 列表没有装满的情况下：
push(x) 的实现：
queue[tail] = x
tail = (tail+1) % capacity
pop() 的实现：
head = (head+1) % capacity
capacity 可以是 4,8,16.....
3) 如何判断队列是否为空：
方法 1：维护一个元素总数 size，size == 0 即为空
方法 2：不维护 size，浪费 queue 中一个单元的存储空间
head == tail 即为空
4) 如何判断队列是否为满:
方法 1：维护一个元素总数 size，size == capacity 即为满
方法 2：不维护 size, 浪费 queue 中一个单元的存储空间，
(tail + 1) % capacity == head 即为满
如果不浪费，就无法区分 head == tail 是队列空导致，还是队列满导致
5) 若一个 push 操作后导致列表满：
i. 建一个大小是原列表 k 倍大的新列表 ($k > 1$, 可以取 1.5,2.....)
ii. 将原列表内容全部拷贝到新列表，作为新队列
iii. 重新设置新列表的 head 和 tail
iv. 原列表空间自动被 Python 解释器回收

导致队列满的 push 的时间复杂度是 $O(n)$。平均 push 操作是 $O(1)$
Python 列表 append 做到 $O(1)$ 的实现也是这种原理, 且 k 取 1.125，空间换时间
若每次增加空间只增加固定数量，比如 20 个单元，则 push 平均复杂度还是 $O(n)$

```
class Queue:
    _initC = 8    #存放队列的列表的初始容量
    _expandFactor = 1.5 #扩充容量时容量增加的倍数
    def __init__(self):
        self._q = [None for i in range(Queue._initC)]
        self._size = 0              #队列元素个数
        self._capacity = Queue._initC #队列最大容量
        self._head = self._rear = 0
    def isEmpty(self):
        return self._size == 0
    def front(self): #看队头元素。空队列导致RE
        if self._size == 0:
            raise Exception("Queue is empty")
        return self._q[self._head]
    def back(self): #看队尾元素，空队列导致RE
        if self._size == 0:
            raise Exception("Queue is empty")
        if self._rear > 0:
            return self._q[self._rear - 1]
        else:
            return self._q[-1]
    def push(self,x):
        if self._size == self._capacity:
            tmp = [None for i in range(
                  int(self._capacity*Queue._expandFactor
                      ))]
            k = 0
            while k < self._size:
                tmp[k] = self._q[self._head]
                self._head = (self._head + 1) % self.
                    _capacity
                k += 1
            self._q = tmp #原来SELF._Q的空间会被PYTHON自动
                释放
            self._q[k] = x
            self._head,self._rear = 0,k+1
            self._capacity = int(
                  self._capacity*Queue._expandFactor)
        else:
            self._q[self._rear] = x
            self._rear = (self._rear + 1) % self.
                _capacity
        self._size += 1
    def pop(self):
        if self._size == 0:
            raise Exception("Queue is empty")
        self._size -= 1
        self._head = (self._head + 1) % len(self._q)

q = Queue()
for i in range(1,314):
```

```
    q.push(i)
    print(q.back(),end=",")
print()
while not q.isEmpty():
    print(q.front(),end=",")
    q.pop()
```

6 Math vs. text vs. functions

In properly typeset mathematics variables appear in italics (e.g., $f(x) = x^2 + 2x - 3$). The exception to this rule is predefined functions (e.g., $\sin(x)$). Thus it is important to **always** treat text, variables, and functions correctly. See the difference between x and x, -1 and -1, and $sin(x)$ and $\sin(x)$.

There are two ways to present a mathematical expression— *inline* or as an *equation*.

6.1 Inline mathematical expressions

Inline expressions occur in the middle of a sentence. To produce an inline expression, place the math expression between dollar signs (`$`). For example, typing `$90^{\circ}$ is the same as $\frac{\pi}{2}$ radians` yields 90° is the same as $\frac{\pi}{2}$ radians.

6.2 Equations

Equations are mathematical expressions that are given their own line and are centered on the page. These are usually used for important equations that deserve to be showcased on their own line or for large equations that cannot fit inline. To produce an inline expression, place the mathematical expression between the symbols `\[` and `\]`. Typing `\[x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}\]` yields

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

6.3 Displaystyle

To get full-sized inline mathematical expressions use `\displaystyle`. Use this sparingly. Typing `I want this $\displaystyle \sum_{n=1}^{\infty} \frac{1}{n}$, not this $\sum_{n=1}^{\infty} \frac{1}{n}$.` yields

I want this $\displaystyle \sum_{n=1}^{\infty} \frac{1}{n}$, not this $\sum_{n=1}^{\infty} \frac{1}{n}$.

7 Images

You can put images (pdf, png, jpg, or gif) in your document. They need to be in the same location as your .tex file when you compile the document. Omit `[width=.5in]` if you want the image to be full-sized.

```
\begin{figure}[ht]
\includegraphics[width=.5in]{imagename.jpg}
\caption{The (optional) caption goes here.}
\end{figure}
```

7.1 Text decorations

Your text can be *italics* (`\textit{italics}`), **boldface** (`\textbf{boldface}`), or underlined (`\underline{underlined}`).

Your math can contain boldface, $\mathbf{R}$ (`\mathbf{R}`), or blackboard bold, $\mathbb{R}$ (`\mathbb{R}`). You may want to used these to express the sets of real numbers ($\mathbb{R}$ or **R**), integers ($\mathbb{Z}$ or **Z**), rational numbers ($\mathbb{Q}$ or **Q**), and natural numbers ($\mathbb{N}$ or **N**).

To have text appear in a math expression use `\text`. `(0,1]=\{x\in\mathbb{R}:x>0\text{ and }x\le 1\}` yields $(0,1] = \{x \in \mathbb{R} : x > 0 \text{ and } x \le 1\}$. (Without the `\text` command it treats "and" as three variables: $(0,1] = \{x \in \mathbb{R} : x > 0 and x \le 1\}$.)

8 Spaces and new lines

LaTeX ignores extra spaces and new lines. For example,

```
This   sentence will       look
fine after      it is     compiled.
```

This sentence will look fine after it is compiled.

Leave one full empty line between two paragraphs. Place `\\` at the end of a line to create a new line (but not create a new paragraph).

```
This
compiles

like\\
this.
```

This compiles
like
this.

Use `\noindent` to prevent a paragraph from indenting.

9 Comments

Use `%` to create a comment. Nothing on the line after the `%` will be typeset. `$f(x)=\sin(x)$ %this is the sine function` yields $f(x) = \sin(x)$

10 Delimiters

description	*command*	*output*
parentheses	`(x)`	(x)
brackets	`[x]`	[x]
curly braces	`\{x\}`	{x}

To make your delimiters large enough to fit the content, use them together with `\right` and `\left`. For example,

```
\left\{\sin\left(\frac{1}{n}\right)\right\}_{n}^{\infty}
```

produces

$$\left\{\sin\left(\frac{1}{n}\right)\right\}_{n}^{\infty}.$$

Curly braces are non-printing characters that are used to gather text that has more than one character. Observe the differences between the four expressions `x^2`, `x^{2}`, `x^2t`, `x^{2t}` when typeset: x^2, x^{2}, x^2t, x^{2t}.

11 Lists

You can produce ordered and unordered lists.

description	*command*	*output*

unordered list	`\begin{itemize}`
	`  \item`
	`  Thing 1`
	`  \item`
	`  Thing 2`
	`\end{itemize}`
		• Thing 1
		• Thing 2

ordered list	`\begin{enumerate}`
	`  \item`
	`  Thing 1`
	`  \item`
	`  Thing 2`
	`\end{enumerate}`
		1. Thing 1
		2. Thing 2

12 Symbols (in *math* mode)

12.1 The basics

description	*command*	*output*
addition	`+`	$+$
subtraction	`-`	$-$
plus or minus	`\pm`	$\pm$
multiplication (times)	`\times`	$\times$
multiplication (dot)	`\cdot`	$\cdot$
division symbol	`\div`	$\div$
division (slash)	`/`	$/$
circle plus	`\oplus`	$\oplus$
circle times	`\otimes`	$\otimes$
equal	`=`	$=$
not equal	`\ne`	$\neq$
less than	`<`	$<$
greater than	`>`	$>$
less than or equal to	`\le`	$\leq$
greater than or equal to	`\ge`	$\geq$
approximately equal to	`\approx`	$\approx$
infinity	`\infty`	∞
dots	`1,2,3,\ldots`	$1, 2, 3, \ldots$
dots	`1+2+3+\cdots`	$1 + 2 + 3 + \cdots$
fraction	`\frac{a}{b}`	$\frac{a}{b}$
square root	`\sqrt{x}`	$\sqrt{x}$
nth root	`\sqrt[n]{x}`	$\sqrt[n]{x}$
exponentiation	`a^b`	a^b
subscript	`a_b`	a_b
absolute value	`|x|`	$\|x\|$
natural log	`\ln(x)`	$\ln(x)$
logarithms	`\log_{a}b`	$\log_a b$
exponential function	`e^x=\exp(x)`	$e^x = \exp(x)$
degree	`\deg(f)`	$\deg(f)$

12.2 Functions

description	*command*	*output*
maps to	`\to`	$\to$
composition	`\circ`	$\circ$

piecewise function	`|x|=`
	`\begin{cases}`
	`x & x\ge 0\\`
	`-x & x<0`
	`\end{cases}`
		$\|x\| = \begin{cases} x & x \geq 0 \\ -x & x < 0 \end{cases}$

12.3 Greek and Hebrew letters

command	*output*	*command*	*output*
`\alpha`	α	`\tau`	τ
`\beta`	β	`\theta`	θ
`\chi`	χ	`\upsilon`	υ
`\delta`	δ	`\xi`	ξ
`\epsilon`	ϵ	`\zeta`	ζ
`\varepsilon`	ε	`\Delta`	Δ
`\eta`	η	`\Gamma`	Γ
`\gamma`	γ	`\Lambda`	Λ
`\iota`	ι	`\Omega`	Ω
`\kappa`	κ	`\Phi`	Φ
`\lambda`	λ	`\Pi`	Π
`\mu`	μ	`\Psi`	Ψ
`\nu`	ν	`\Sigma`	Σ
`\omega`	ω	`\Theta`	Θ
`\phi`	ϕ	`\Upsilon`	Υ
`\varphi`	φ	`\Xi`	Ξ
`\pi`	π	`\aleph`	$\aleph$
`\psi`	ψ	`\beth`	$\beth$
`\rho`	ρ	`\daleth`	$\daleth$
`\sigma`	σ	`\gimel`	$\gimel$

12.4 Set theory

description	*command*	*output*
set brackets	`\{1,2,3\}`	$\{1,2,3\}$
element of	`\in`	$\in$
not an element of	`\not\in`	$\notin$
subset of	`\subset`	$\subset$
subset of	`\subseteq`	$\subseteq$
not a subset of	`\not\subset`	$\not\subset$
contains	`\supset`	$\supset$
contains	`\supseteq`	$\supseteq$
union	`\cup`	$\cup$
intersection	`\cap`	$\cap$
big union	`\bigcup_{n=1}^{10}A_n`	$\bigcup_{n=1}^{10} A_n$
big intersection	`\bigcap_{n=1}^{10}A_n`	$\bigcap_{n=1}^{10} A_n$
empty set	`\emptyset`	$\emptyset$
power set	`\mathcal{P}`	$\mathcal{P}$
minimum	`\min`	$\min$
maximum	`\max`	$\max$
supremum	`\sup`	$\sup$
infimum	`\inf`	$\inf$
limit superior	`\limsup`	$\limsup$
limit inferior	`\liminf`	$\liminf$
closure	`\overline{A}`	$\overline{A}$

12.5 Calculus

description	*command*	*output*
derivative	`\frac{df}{dx}`	$\frac{df}{dx}$
derivative	`\f'`	f'
partial derivative	`\frac{\partial f}{\partial x}`	$\frac{\partial f}{\partial x}$
integral	`\int`	$\int$
double integral	`\iint`	$\iint$
triple integral	`\iiint`	$\iiint$
limits	`\lim_{x\to \infty}`	$\lim_{x \to \infty}$
summation	`\sum_{n=1}^{\infty}a_n`	$\sum_{n=1}^{\infty} a_n$
product	`\prod_{n=1}^{\infty}a_n`	$\prod_{n=1}^{\infty} a_n$

12.6 Logic

description	*command*	*output*
not	`\sim`	$\sim$
and	`\land`	$\land$
or	`\lor`	$\lor$
if...then	`\to`	$\to$
if and only if	`\leftrightarrow`	$\leftrightarrow$
logical equivalence	`\equiv`	$\equiv$
therefore	`\therefore`	$\therefore$
there exists	`\exists`	$\exists$
for all	`\forall`	$\forall$
implies	`\Rightarrow`	$\Rightarrow$
equivalent	`\Leftrightarrow`	$\Leftrightarrow$

12.7 Linear algebra

description	*command*	*output*
vector	`\vec{v}`	$\vec{v}$
vector	`\mathbf{v}`	$\mathbf{v}$
norm	`||\vec{v}||`	$\|\|\vec{v}\|\|$
matrix	`\left[ \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6\\ 7 & 8 & 0 \end{array} \right]`	$\left[\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6\\ 7 & 8 & 0 \end{array} \right]$
determinant	`\left| \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{array} \right|`	$\left\| \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{array} \right\|$
determinant	`\det(A)`	$\det(A)$
trace	`\operatorname{tr}(A)`	$\operatorname{tr}(A)$
dimension	`\dim(V)`	$\dim(V)$

12.8 Number theory

description	*command*	*output*
divides	`|`	$\|$
does not divide	`\not |`	$\nmid$
div	`\operatorname{div}`	div
mod	`\mod`	$\mod$
greatest common divisor	`\gcd`	$\gcd$
ceiling	`\lceil x \rceil`	$\lceil x \rceil$
floor	`\lfloor x \rfloor`	$\lfloor x \rfloor$

12.9 Geometry and trigonometry

description	*command*	*output*
angle	`\angle ABC`	$\angle ABC$
degree	`90^{\circ}`	90°
triangle	`\triangle ABC`	$\triangle ABC$
segment	`\overline{AB}`	$\overline{AB}$
sine	`\sin`	$\sin$
cosine	`\cos`	$\cos$
tangent	`\tan`	$\tan$
cotangent	`\cot`	$\cot$
secant	`\sec`	$\sec$
cosecant	`\csc`	$\csc$
inverse sine	`\arcsin`	$\arcsin$
inverse cosine	`\arccos`	$\arccos$
inverse tangent	`\arctan`	$\arctan$

13 Symbols (in *text* mode)

The followign symbols do **not** have to be surrounded by dollar signs.

description	*command*	*output*
dollar sign	`\$`	$
percent	`\%`	%
ampersand	`\&`	&
pound	`\#`	#
backslash	`\textbackslash`	\
left quote marks	``` `` ```	“
right quote marks	`''`	”
single left quote	`` ` ``	‘
single right quote	`'`	’
hyphen	`X-ray`	X-ray
en-dash	`pp. 5--15`	pp. 5–15
em-dash	`Yes---or no?`	Yes—or no?

14 Resources

Great symbol look-up site: Detexify
LaTeX Mathematical Symbols
The Comprehensive LaTeX Symbol List
The Not So Short Introduction to LaTeX 2ε
TUG: The TeX Users Group
CTAN: The Comprehensive TeX Archive Network

LaTeX for the Mac: MacTeX
LaTeX for the PC: TeXnicCenter and MiKTeX
LaTeX online: WriteLaTeX.

Dave Richeson, Dickinson College, http://divisbyzero.com/