Heap Sort
Heapsort Algorithm

Heapsort(A):

#Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length downto 2
    discard node i from heap (decrement heap size)
    Max-heapify(A, 1) because new root may violate max heap property
Build Max Heap

Build_Max_Heap(A):
    set heap size to the length of the array
    iterate j from \([A.\text{length}/2]\) down to 1:
        Max-heapify(A, j)
The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

```python
def parent(i):
    return i//2

def left(i):
    try:
        return 2*i
    except:
        pass

def right(i):
    try:
        return 2*i + 1
    except:
        pass
```
Max-Heapify

```python
def max_heapify(arr, i):
    n = len(arr)
    l = left(i)
    r = right(i)

    if l <= n and arr[l] > arr[i]:
        largest = l
    else:
        largest = i

    if r <= n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        temp = arr[i]
        arr[i] = arr[largest]
        arr[largest] = temp
        max_heapify(arr, largest)

    return arr
```
Start with an array (it is not a max heap)
Build_Max_Heap(A):
    set heap size to the length of the array
    iterate j from \([A.length/2]\) down to 1:
        Max-heapify(A, j)

def max_heapify(arr, i):
    n = len(arr)
    l = left(i)
    r = right(i)
    if l <= n and arr[l] > arr[i]:
        largest = l
    else:
        largest = i
    if r <= n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        temp = arr[i]
        arr[i] = arr[largest]
        arr[largest] = temp
        max_heapify(arr, largest)
    return arr

def parent(i):
    return i/2

def left(i):
    try:
        return 2*i
    except:
        pass

def right(i):
    try:
        return 2*i + 1
    except:
        pass
Build_Max_Heap(A):
    set heap size to the length of the array
    iterate j from \([A.length/2]\) down to 1:
        Max-heapify(A, j)
Exchange 9 and 1

Build_Max_Heap(A):
   set heap size to the length of the array
   iterate j from \([A.length/2]\) down to 1:
      Max-heapify(A, j)
Build_Max_Heap(A):
    set heap size to the length of the array
    iterate j from \[A.length/2\] down to 1:
        Max-heapify(A, j)
Exchange 6 and 11

Build_Max_Heap(A):
  set heap size to the length of the array
  iterate j from \([A.length/2]\) down to 1:
    Max-heapify(A, j)

```
def max_heapify(arr, i):
    n = len(arr)
    l = left(i)
    r = right(i)
    if l <= n and arr[l] > arr[i]:
        largest = l
    else:
        largest = i
    if r <= n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        temp = arr[i]
        arr[i] = arr[largest]
        arr[largest] = temp
    max_heapify(arr, largest)
return arr
```
Exchange 6 and 10

Build_Max_Heap(A):
   set heap size to the length of the array
   iterate j from [A.length/2] down to 1:
      Max-heapify(A, j)

```
def max_heapify(arr, i):
    n = len(arr)
    l = left(i)
    r = right(i)
    if l <= n and arr[l] > arr[i]:
        largest = l
    else:
        largest = i
    if r <= n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        temp = arr[i]
        arr[i] = arr[largest]
        arr[largest] = temp
    max_heapify(arr, largest)
return arr
```
Exchange 6 and 7

Build_Max_Heap(A):
    set heap size to the length of the array
    iterate j from [A.length/2] down to 1:
        Max-heapify(A, j)

def max_heapify(arr, i):
    n = len(arr)
    l = left(i)
    r = right(i)
    if l <= n and arr[l] > arr[i]:
        largest = l
    else:
        largest = i
    if r <= n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        temp = arr[i]
        arr[i] = arr[largest]
        arr[largest] = temp
        max_heapify(arr, largest)
    return arr
max_heapify

Build_Max_Heap(A):
    set heap size to the length of the array
    iterate j from [A.length/2] down to 1:
        Max-heapify(A, j)

```python
def max_heapify(arr, i):
    n = len(arr)
    l = left(i)
    r = right(i)
    if l <= n and arr[l] > arr[i]:
        largest = l
    else:
        largest = i
    if r <= n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        temp = arr[i]
        arr[i] = arr[largest]
        arr[largest] = temp
        max_heapify(arr, largest)
    return arr
```

11 10 9 8 7 1 3 2 4 6

j i largest
Heapsort(A):

#Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length downto 2
  discard node i from heap (decrement heap size)
  Max-heapify(A,1) because new root may violate max heap property
Exchange 11 and 6

Heapsort(A):

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length downto 2
  discard node i from heap (decrement heap size)
  Max-heapify(A, 1) because new root may violate max heap property
Remove 11 from the heap
Swap 6 and 10
Exchange 6 and 8
Exchange 10 and 4

Heapsort(A):

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length downto 2
   discard node i from heap (decrement heap size)
Max-heapify(A, 1) because new root may violate max heap property
Remove 10 from the heap
Exchange 4 and 9
Heapsort(A):

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length downto 2
   discard node i from heap (decrement heap size)
Max-heapify(A, 1) because new root may violate max heap property

Exchange 9 and 2
Remove 9 from the heap
Exchange 2 and 8
Exchange 2 and 7

8

2

6

7

1

3

4

9

10

11

8 2 4 6 7 1 3 9 10 11
Exchange 8 and 3

Heapsort(A):

- Create max heap
- Build_Max_Heap from unordered array A
- Finish sorting
  - Iterate i from A.length downto 2
    - Discard node i from heap (decrement heap size)
    - Max-heapify(A, 1) because new root may violate max heap property

Diagram:

```
  8
 /|
7 4
/ \|
6 2 1
   /\|
  3 9 10 11
```

Table:

<table>
<thead>
<tr>
<th>8</th>
<th>7</th>
<th>4</th>
<th>6</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
</table>
Remove 8 from the heap
Exchange 3 and 7
Exchange 3 and 6
Heapsort(A):

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length downto 2
discard node i from heap (decrement heap size)
Max-heapify(A, 1) because new root may violate max heap property
Remove 7 from the heap
Exchange 1 and 6
Exchange 1 and 3
Exchange 6 and 2 and remove from the heap

Heapsort(A):

1. Create max heap
2. Build_Max_Heap from unordered array A
3. Finish sorting
4. Iterate i from A.length downto 2
6. Discard node i from heap (decrement heap size)
7. Max-heapify(A, 1) because new root may violate max heap property
Exchange 4 and 2
Heapsort(A):

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length downto 2

discard node i from heap (decrement heap size)
Max-heapify(A, 1) because new root may violate max heap property
Remove 4, exchange 1 and 3
Heapsort(A):

1. Create max heap
   - Build_Max_Heap from unordered array A

2. Finish sorting
   - Iterate i from A.length downto 2
     - Discard node i from heap (decrement heap size)
     - Max-heapify(A, 1) because new root may violate max heap property

Exchange 2 and 3, and remove 3 from heap.
Heapsort(A):

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length down to 2

discard node i from heap (decrement heap size)
Max-heapify(A, 1) because new root may violate max heap property
The array is sorted

```
1 2 3 4 6 7 8 9 10 11
```
Sorted Output

Heap Sort Algorithm (build + sort)

11 10 9 8 7 1 3 2 4 6

Build Heap (Max)

Sort Max Heap

1 2 3 4 6 7 8 9 10 11
Heapsort Algorithm

Heapsort(A):

#Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
iterate i from A.length downto 2

discard node i from heap (decrement heap size)

Max-heapify(A, 1) because new root may violate max heap property
Build Max Heap

Build_Max_Heap(A):
    set heap size to the length of the array
    iterate j from [A.length/2] down to 1:
        Max-heapify(A, j)
The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

```python
def parent(i):
    return i/2

def left(i):
    try:
        return 2*i
    except:
        pass

def right(i):
    try:
        return 2*i + 1
    except:
        pass
```
def max_heapify(arr, i):
    n = len(arr)
    l = left(i)
    r = right(i)

    if l <= n and arr[l] > arr[i]:
        largest = l
    else:
        largest = i

    if r <= n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        temp = arr[i]
        arr[i] = arr[largest]
        arr[largest] = temp
        max_heapify(arr, largest)

    return arr