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 \([A.length/2]\) 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) - 1
    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, n, 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)

\[
def \text{parent}(i):
    return i/2
\]

\[
def \text{left}(i):
    try:
    return 2*i
    except:
    pass
\]

\[
def \text{right}(i):
    try:
    return 2 *i + 1
    except:
    pass
\]

\[
def \text{max_heapify}(arr,i):
    n = len(arr) - 1
    l = \text{left}(i)
    r = \text{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
        \text{max_heapify}(arr,largest)
    return arr
\]
Exchange 4 and 8

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)
Exchange 10 and 11

Build_MAX_Heap(A):
   set heap size to the length of the array
   iterate \( j \) from \( \lfloor A.length/2 \rfloor \) 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)

def max_heapify(arr, i):
    n = len(arr) - 1
    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) - 1
    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
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) - 1
    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)

def max_heapify(arr,i):
    n = len(arr) - 1
    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
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
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
Remove 11 from the heap

```
  6
 / \
10  9
/  / \  /
8  7 13 2
/   /   /
2   4   11
```

6 10 9 8 7 1 3 2 4 11
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
Exchange 9 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 9 from the heap

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

Exchange 6 and 2 and remove from the heap

```
6  3  4  1  2
7  8  9 10 11
```
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
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 downto 2
   discard node i from heap (decrement heap size)
Max-heapify(A, 1) because new root may violate max heap property
Exchange 1 and 2 and remove from heap

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
The array is sorted

1

2

4

6

7

8

9

10

11

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

Heap Sort Algorithm (build + sort)

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)
Heap

- 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

def max_heapify(arr, i):
    n = len(arr) - 1
    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, n, largest)

return arr