به نام خدا

الگوریتم های مرتب سازی:

الگوریتم Quick Sort:

الگوریتم مرتب سازی سریع از نوع الگوریتم های مقایسه‌ای است و تکنیک تقسیم و غلبه برای مرتب سازی استفاده می‌کند.

مراحل الگوریتم quick sort:

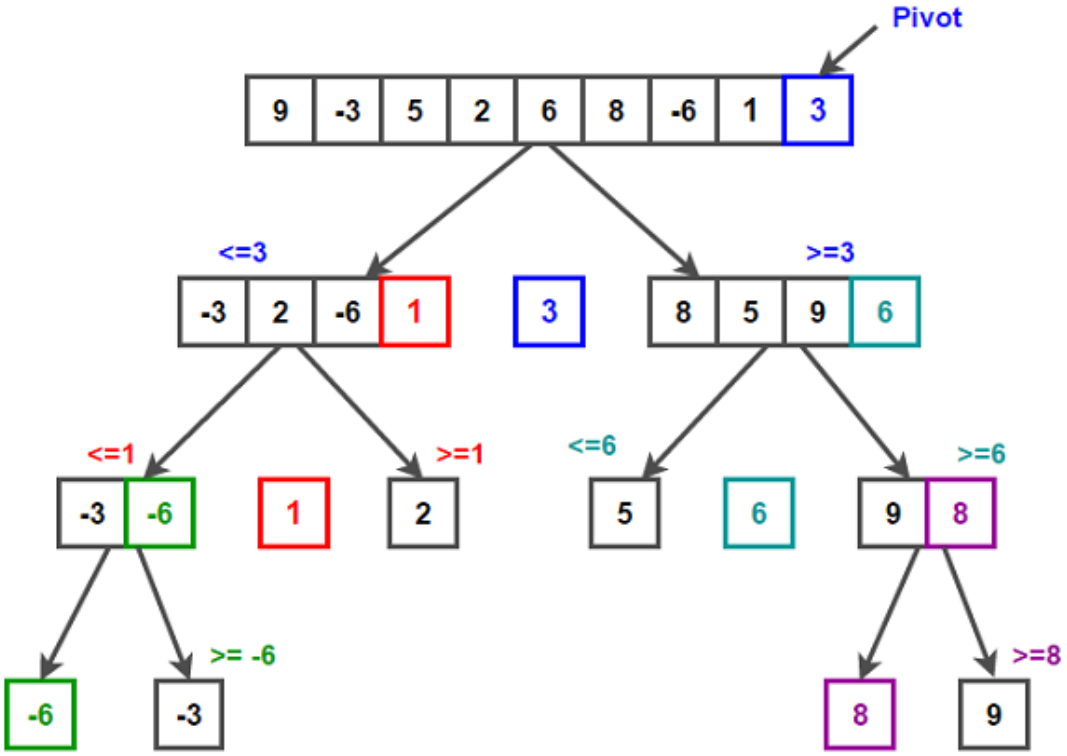
۱. **انتخاب یک عنصر محوری:**
اولین مرحله انتخاب یک عنصر محوری از آرایه است. این عنصر به عنوان یک نقطه مرجع برای تقسیم آرایه به دو قسمت استفاده می شود.

۲. **پارتیشن بندی آرایه:**
مرحله بعدی این است که آرایه را به دو قسمت تقسیم می شود، یکی شامل عناصر کمتر از عنصر محوری و دیگری حاوی عناصر بزرگتر از عنصر محوری است. برای هر عنصر بررسی می شود که آیا کوچکتر یا بزرگتر از عنصر محوری است. اگر کمتر از پیوت باشد در پارتیشن سمت چپ و اگر بزرگتر از پیوت باشد در پارتیشن سمت راست قرار می گیرد.

۳. **مرتب سازی بازگشتی پارتیشن ها:**
بعد از پارتیشن بندی ارایه، الگوریتم به صورت بازگشتی همان فرآیند را برای پارتیشن های چپ و راست اعمال می کند. این کار تا زمانی ادامه می‌یابد که پارتیشن‌ها فقط یک عنصر داشته باشند، در این مرحله الگوریتم، آرایه مرتب شده را برمی‌گرداند.

۴. **ترکیب پارتیشن ها:**

پس از مرتب شدن پارتیشن های چپ و راست، الگوریتم آنها را ترکیب می کند تا آرایه مرتب شده نهایی را تولید کند.



الگوریتم quick sort

مزایا و معایب الگوریتم Quick Sort:

مزایا:

- **سرعت**: quicksort دارای میانگین پیچیدگی زمانی O(n log n) است که آن را به یکی از سریع ترین الگوریتم های مرتب سازی به طور متوسط تبدیل می کند. این بدان معناست که برای مرتب سازی مجموعه داده های بزرگ مناسب است.

- **استفاده کارآمد از حافظه**: یک الگوریتم درجا است، به این معنی که داده ها را بدون نیاز به حافظه اضافی برای ذخیره سازی موقت، درون همان آرایه مرتب می کند. این باعث می شود در سناریوهایی با حافظه محدود مفید باشد.
- **پیاده سازی ساده:** مفهوم کلی تقسیم بندی آرایه حول یک عنصر محوری نسبتا ساده است و درک و پیاده سازی آن را در مقایسه با دیگر از الگوریتم های مرتب سازی آسان تر می کند.
- **قابل انطباق**: در حالی که به طور ذاتی تطبیقی نیست، بهینه‌سازی‌هایی مانند انتخاب تصادفی محور می‌توانند عملکرد آن را در شرایط مختلف به طور قابل توجهی بهبود بخشد.

معایب:

- **بدترین حالت پیچیدگی زمانی (worst case)**: در بدترین حالت، که در آن عنصر محوری به طور مداوم آرایه را به طور ناهموار تقسیم می کند، پیچیدگی زمانی می تواند به O(n^2) کاهش یابد که آن را به طور قابل توجهی کندتر از عملکرد متوسط آن می کند.
- **ناپایداری**: quicksort ترتیب اصلی عناصر برابر را حفظ نمی کند، که ممکن است در برنامه های خاص بحرانی باشد. اگر ترتیب مهم است از یک الگوریتم مرتب سازی پایدار مانند merge sort استفاده می شود.
- **حساسیت نسبت به انتخاب پیوت**: عملکرد تا حد زیادی به عنصر محوری انتخاب شده بستگی دارد. انتخاب ضعیف می تواند منجر به بدترین حالت شود. تصادفی کردن انتخاب پینوت می تواند این مشکل را کاهش دهد اما پیچیدگی را افزایش می دهد.
- **با لیست های پیوندی خوب کار نمی کند:** quicksort در لیست های پیوندی و سایر ساختارهای مبتنی بر اشاره گر ضعیف عمل می کند.

کد الگوریتم quick sort در پایتون:

```
def partition(array, low, high):

  # choose the rightmost element as pivot
  pivot = array[high]

  # pointer for greater element
  i = low - 1

  # traverse through all elements
  # compare each element with pivot
  for j in range(low, high):
    if array[j] <= pivot:
      # if element smaller than pivot is found
      # swap it with the greater element pointed by i
      i = i + 1

      # swapping element at i with element at j
      (array[i], array[j]) = (array[j], array[i])

  # swap the pivot element with the greater element specified by i
  (array[i + 1], array[high]) = (array[high], array[i + 1])

  return i + 1

def quickSort(array, low, high):
  if low < high:

    # find pivot element such that
    # element smaller than pivot are on the left
    # element greater than pivot are on the right
    pi = partition(array, low, high)

    # recursive call on the left of pivot
    quickSort(array, low, pi - 1)

    # recursive call on the right of pivot
    quickSort(array, pi + 1, high)
```

الگوریتم Heap Sort:

الگوریتم مرتب سازی هرمی از ساختار داده‌های هرمی برای سازمان دهی عناصر یک آرایه استفاده می‌کند تا بتوان آنها را مرتب کرد. هرم نوع خاصی از ساختار داده مبتنی بر درخت است وگره والد همیشه بیشتر از (یا کمتر از) گره های فرزند خود است.

ایده اصلی مرتب سازی هرمی این است که ابتدا یک هرم از آرایه مرتب نشده ساخته می شود. سپس الگوریتم به طور مکرر بزرگترین (یا کوچکترین) عنصر را از هرم حذف می کند، آن را در انتهای (یا ابتدای) آرایه مرتب شده قرار می دهد و سپس عناصر باقیمانده را مجدداً برای حفظ خاصیت هرمی پر می کند. این روند تا مرتب شدن کل آرایه ادامه می یابد. پیچیدگی زمانی مرتب سازی هرمی O(n log n) است. مرتب سازی هرمی یک الگوریتم مرتب سازی پایدار است، به این معنی که ترتیب عناصر برابر حفظ می‌شود.

مراحل الگوریتم مرتب سازی هرمی:

۱. ساخت هرم:

کار با ساختن یک هرم از آرایه مرتب نشده شروع می شود. این کار با تقسیم مکرر آرایه به زیرآرایه های کوچکتر و سپس انتخاب بزرگترین (یا کوچکترین) عنصر از هر زیرآرایه برای تشکیل ریشه یک پشته جدید انجام می شود. این روند تا زمانی ادامه می یابد که کل آرایه به عناصر جداگانه تقسیم شود، در این مرحله الگوریتم به مرحله بعدی می رود.

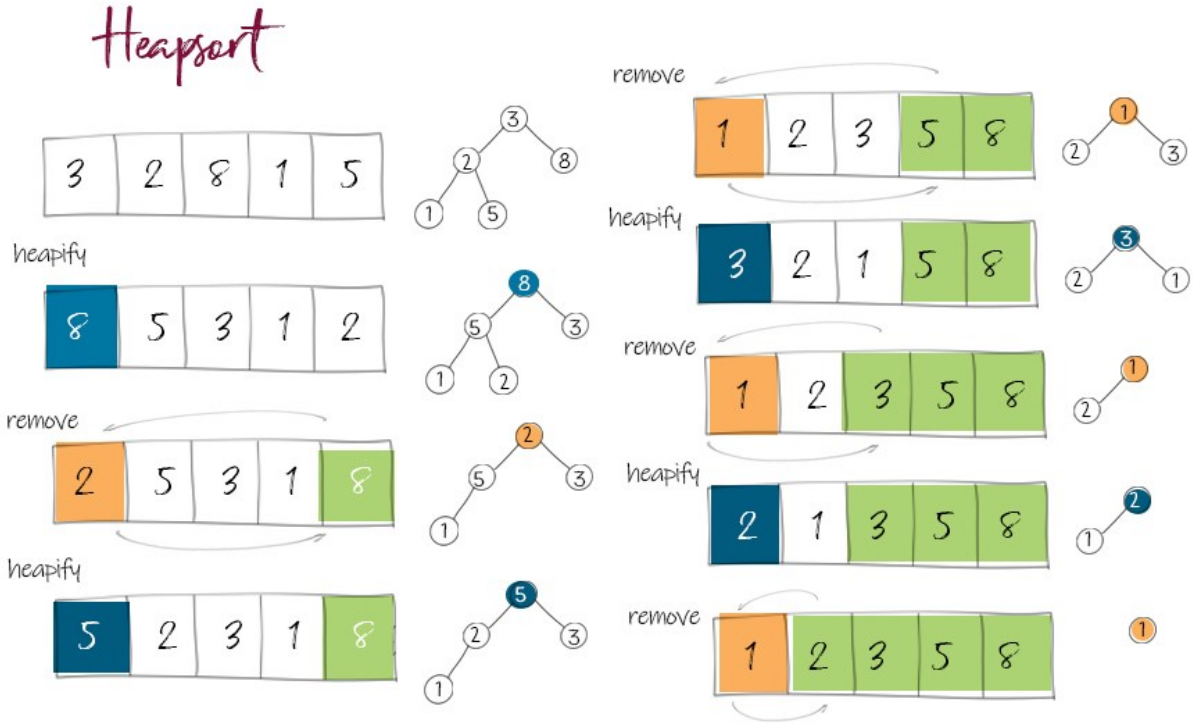
۲. انباشته کردن:

هنگامی که هرم ساخته شد، الگوریتم به طور مکرر بزرگترین (یا کوچکترین) عنصر را از هرم حذف می کند، آن را در انتهای (یا ابتدای) آرایه مرتب شده قرار می دهد و سپس عناصر باقی مانده را مجدداً برای حفظ هرم مجدد به صورت هرمی انباشت می کند. این روند تا مرتب شدن کل آرایه ادامه می یابد.

۳. دوباره انباشته کردن:

پس از حذف هر عنصر از هرم، الگوریتم برای حفظ خاصیت هرمی، عناصر باقی مانده را دوباره انباشت میکند. این شامل تقسیم مکرر عناصر باقیمانده به زیرآرایه های کوچکتر و سپس انتخاب بزرگترین (یا کوچکترین)

عنصر از هر زیرآرایه برای تشکیل ریشه یک هرم جدید است. این روند تا مرتب شدن کل آرایه ادامه می یابد.
الگوریتم مراحل ۲ و ۳ را تکرار می کند تا کل آرایه مرتب شود.



الگوریتم heap sort

مزایا و معایب الگوریتم Heap Sort:

مزایا:

- **کارایی**: مرتب سازی هرمی دارای پیچیدگی زمانی O(n log n) است که برای الگوریتم‌های مرتب سازی بهینه در نظر گرفته می‌شود. این بدان معناست که سرعت مرتب سازی آن با مجموعه داده های بزرگ به خوبی مقیاس می شود.

- **سادگی**: در مقایسه با سایر الگوریتم های کارآمد مانند مرتب‌سازی سریع، مرتب سازی هرمی برای درک و پیاده سازی آسان‌تر است، و آن را به انتخاب خوبی برای مبتدیان تبدیل می‌کند.

- **مرتب‌سازی درجا:** مرتب سازی هرمی به حداقل حافظه اضافی نیاز دارد O(1) که باعث می‌شود در مقایسه با الگوریتم‌هایی مانند مرتب سازی ادغامی که به فضای اضافی حافظه نیاز دارند کارآمد باشد.

- **پیچیدگی تضمین شده O(n log n)**: برخلاف مرتب سازی سریع، که می تواند در بدترین حالت پیچیدگی O(n^2) داشته باشد، مرتب سازی هرمی به طور مداوم در زمان O(n log n) انجام می شود و عملکرد آن را قابل پیش بینی تر می کند.

معایب:

- اگرچه کارآمد است، مرتب سازی هرمی همیشه سریع‌ترین گزینه نیست. Quicksort اغلب در عمل از آن بهتر عمل می کند.

- مرتب سازی هرمی روی داده‌های تا حدی مرتب شده یا داده های تکراری زیاد کارایی خوبی ندارد، جایی که الگوریتم های دیگری مانند مرتب سازی سریع می‌توانند از ترتیب موجود استفاده کنند.

- مرتب سازی هرمی پایدار نیست، به این معنی که ممکن است ترتیب نسبی عناصر با مقادیر برابر را تغییر دهد. اگر ترتیب چنین عناصری مهم باشد، این می تواند مشکل ساز باشد.

- مرتب سازی هرمی هنگام کار با داده های بسیار پیچیده خیلی کارآمد نیست.

کد الگوریتم **heap sort** در پایتون:

```
def heapify(arr, n, i):
    # Find largest among root and children
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

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

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

    # If root is not largest, swap with largest and continue heapifying
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n//2, -1, -1):
        heapify(arr, n, i)

    for i in range(n-1, 0, -1):
        # Swap
        arr[i], arr[0] = arr[0], arr[i]

        # Heapify root element
        heapify(arr, i, 0)
```

الگوریتم Merge Sort:

مرتب سازی ادغامی یکی از روش های مرتب سازی مبتنی بر تقسیم و غلبه (Divide & Conqure) است. مسئله را به زیر مسئله های کوچکتر تبدیل می‌کند و سپس آن را حل می‌کند.

مراحل الگوریتم مرتب سازی ادغامی:

۱. ابتدا بررسی می کند که آیا آرایه ورودی یک یا صفر عنصر دارد، اگر چنین بود ارایه مرتب شده است.

۲. آرایه ورودی را به دو نیمه تقسیم می کند.

۳. به صورت بازگشتی الگوریتم مرتب سازی ادغامی را برای هر نیمه اعمال می کند و آنها را جداگانه مرتب می کند.

۴. دو نیمه مرتب شده دوباره با هم در یک آرایه ادغام می شوند.

مزایا و معایب الگوریتم merge sort:

مزایا:

- **پیچیدگی زمانی:**

مرتب سازی ادغامی دارای پیچیدگی زمانی $O(n \log n)$ در همه موارد (بدترین، متوسط و بهترین) است که آن را برای مجموعه داده های بزرگ بسیار کارآمد می کند. این بدان معناست که زمان مرتب سازی متناسب با لگاریتم اندازه داده رشد می‌کند، صرف نظر از اینکه داده ها در ابتدا چگونه مرتب شده‌اند.

- **پایداری:**

مرتب سازی ادغامی یک الگوریتم مرتب سازی پایدار است، به این معنی که ترتیب عناصر برابر را حفظ می‌کند. این می تواند در برنامه های خاصی که حفظ موقعیت های نسبی بسیار حیاتی است، مهم باشد.

- **رویکرد تقسیم و غلبه:**

از روش تقسیم و غلبه استفاده می‌کند که موازی سازی آن را ساده می‌کند و باعث عملکرد کارامد در سیستم های چند هسته‌ای می شود.

- **اجرا:**

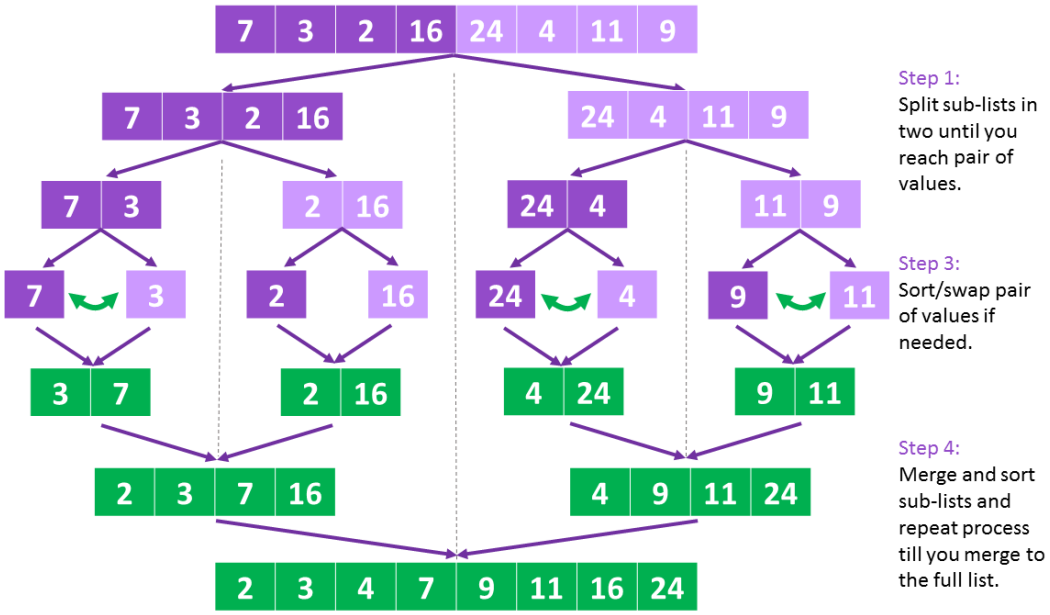
زمان اجرای ثابتی دارد، بیت های مختلف را با زمان‌های مشابه در یک مرحله اجرا می‌کند.

معایب:

- برخلاف الگوریتم های مرتب سازی در جا، مرتب سازی ادغامی به حافظه اضافی برای ذخیره آرایه های فرعی موقت در طول فرآیند ادغام نیاز دارد. این می تواند برای دستگاه هایی با منابع حافظه محدود مشکل ساز باشد.

- به دلیل ماهیت بازگشتی و استفاده اضافی از حافظه، مرتب سازی ادغامی ممکن است کندتر از الگوریتم های مرتب سازی ساده‌تر مانند مرتب سازی درجی برای مجموعه داده های بسیار کوچک باشد.

- در مقایسه با دیگر الگوریتم های مرتب سازی، مرتب سازی ادغامی نیاز به درک بازگشت و رویکرد تقسیم و غلبه دارد که پیاده سازی آن را کمی پیچیده‌تر می‌کند.



الگوریتم merge sort

کد الگوریتم merge sort در پایتون:

```
def mergeSort(array):
   if len(array) > 1:

      #  r is the point where the array is divided into two subarrays
      r = len(array)//2
      L = array[:r]
      M = array[r:]

      # Sort the two halves
      mergeSort(L)
      mergeSort(M)

      i = j = k = 0

      while i < len(L) and j < len(M):
         if L[i] < M[j]:
            array[k] = L[i]
            i += 1
         else:
            array[k] = M[j]
```

```
        j += 1
    k += 1

while i < len(L):
    array[k] = L[i]
    i += 1
    k += 1

while j < len(M):
    array[k] = M[j]
    j += 1
    k += 1
```

مقایسه time complexity الگوریتم های merge sort, quick sort, heap sort: