تمرین سوم مدرسه متخصص دانشمد داده

نامپای- الگوریتم‌های مرتب سازی

استاد

دکتر مسعود کاویانی

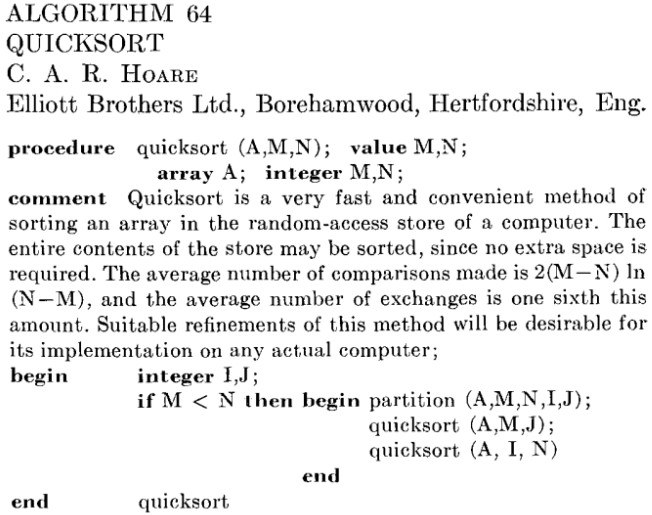
به کوشش: عباس سیف‌السادات

زمستان ۱۴۰۲

فهرست مطالب

توضیح الگوریتم

- Quick sort یک الگوریتم مرتب سازی مشهور است که بر مبنای تقسیم و غلبه (divide-and-conquer) عمل می‌کند.
- این الگوریتم با انتخاب یک عضو محوری (pivot) از آرایه ورودی و پارتیشن بندی سایر عضوها به دو زیرآرایه کار می‌کند.



ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure  quicksort (A,M,N); value M,N;
             array A; integer M,N;
comment  Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin      integer I,J;
           if M < N then begin partition (A,M,N,I,J);
                               quicksort (A,M,J);
                               quicksort (A, I, N)
                         end
end        quicksort
```

این الگوریتم توسط تونی هور (Tony Hoare) در سال 1959 توسعه داده شد. او این الگوریتم را به صورت مقاله‌ای با عنوان "Algorithm 64: Quicksort" در Communications of ACM در سال 1961 منتشر کرد.

https://dl.acm.org/doi/pdf/10.1145/366622.366644

توضیح الگوریتم

رویکردهای مختلف انتخاب pivot

- انتخاب عضو اول آرایه‌ها
- انتخاب عضو آخر آرایه‌ها
- انتخاب عضو وسط آرایه‌ها
- انتخاب به صورت رندوم

```
np.random.choice(array)
```


https://www.geeksforgeeks.org/python-program-for-quicksort/

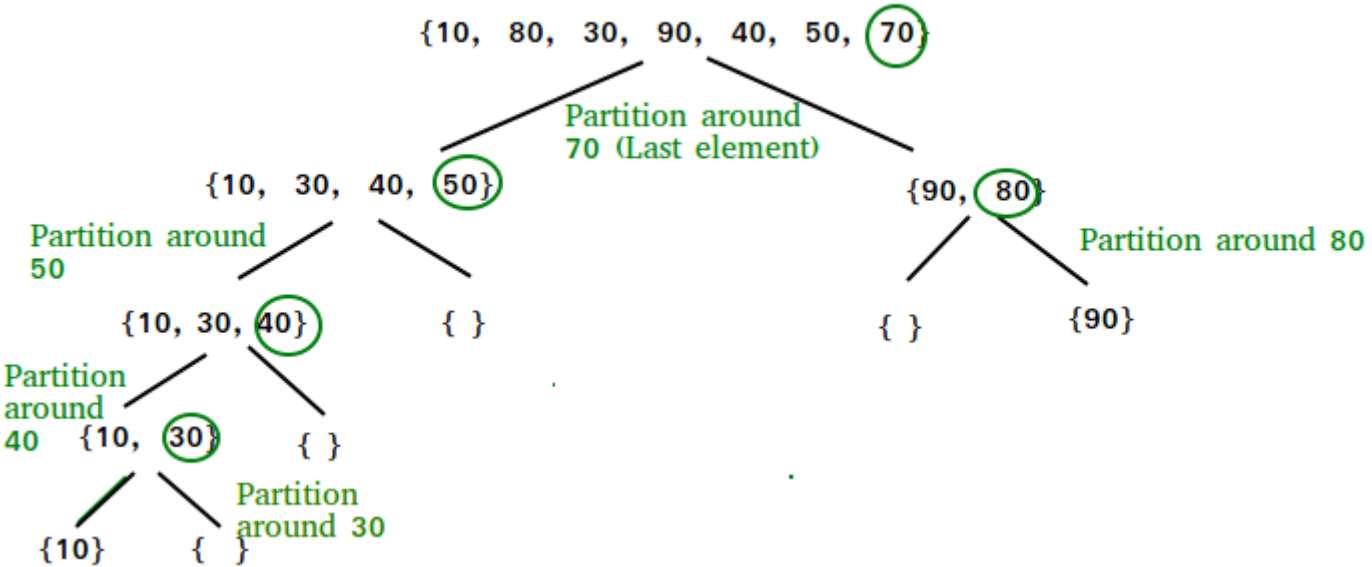
نحوه کارکرد الگوریتم

- این الگوریتم دارای یک آرگمان ورودی است که همان آرایه مدنظر ما جهت مرتب‌سازی است.
- اگر این آرایه عضوی نداشته باشد یا یک عضو داشته باشد؛ همان آرایه ورودی را به خروجی برمی گرداند.
- در غیر اینصورت، یک عضو به عنوان pivot انتخاب می شود.
- سایر عضوها بر این اساس که کوچکتر یا بزرگتر از pivot هستند به دو زیرآرایه راست و چپ تقسیم می‌شوند.
- در گام بعدی همین روال برای هر کدام از زیرآرایه‌ها به صورت بازگشتی تکرار می‌شود تا نهایتا هر کدام از عناصر موجود به صورت آرایه هیچ یا تک عضوی درآیند. در گام آخر کافی است همه این عناصر را در کنار هم قرار دهیم تا آرایه به صورت مرتب شده در خروجی ظاهر شود.

```
np.concatenate([left, np.array([pivot]), right])
```

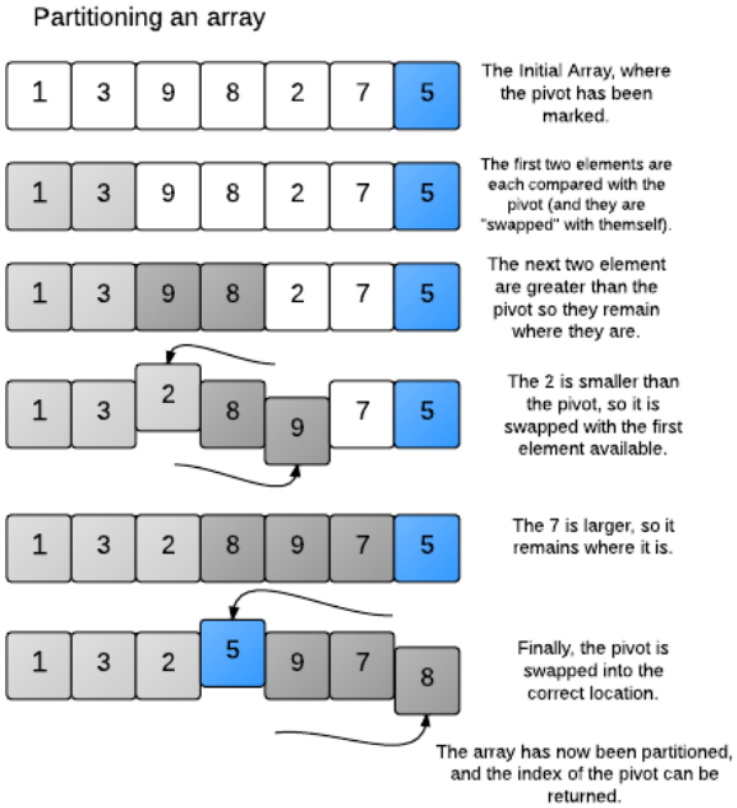

https://www.programiz.com/dsa/quick-sort

مثالی از عملکرد الگوریتم



https://www.geeksforgeeks.org/quick-sort/

نحوه پارتیشن بندی



- انتخاب pivot (در این مثال عنصر آخر)
- مقداردهی partition index به عنوان اولین عنصر آرایه
- سپس مقایسه تک تک عناصر درون index با pivot از ابتدای آرایه؛ اگر عنصر از pivot کمتر باشد، index را اضافه می‌کنیم و به سراغ مقایسه عنصر بعد می‌رویم؛ اگر عنصر بزرگتر از pivot باشد، index را اضافه می‌کنیم و با اولین عنصری که از pivot کمتر است جابه‌جا می‌کنیم؛ این مقایسه را با افزایش index ادامه می‌دهیم تا اینکه همه عناصر کوچکتر از pivot و نهایتا خود pivot با عناصر بزرگتر جابجا شوند.
- در گام بعد همین عملیات را برای دو زیرآرایه حاصل در دو سمت pivot تکرار می‌کنیم.

https://www.codingeek.com/algorithms/quick-sort-algorithm-explanation-implementation-and-complexity/

مزایا و معایب

Pros

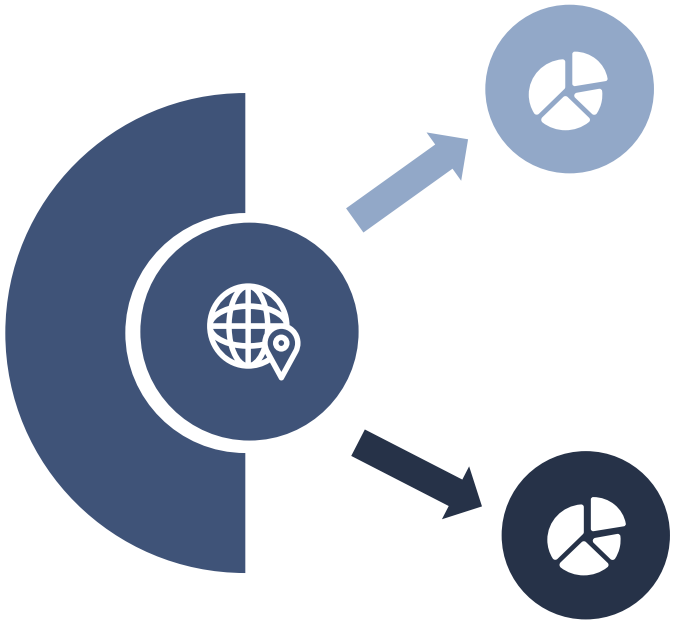
- عملکرد سریع در average-case
- In-place sorting: به این معنا که به حافظه اضافی نیاز ندارد
- عملکرد خوب برای لیست‌های بزرگ
- سادگی در فهم و استفاده در کاربردهای مختلف
- حداقل تعداد مقایسه‌ها
- Cache-friendly

Cons

- در صورتی که انتخاب pivot مناسب نباشد پیچیدگی زمانی در بدترین حالت خود قرار می‌گیرد $O(n^2)$
- عدم پایداری؛ ممکن است ترتیب نسبی عناصر مساوی را تغییر دهد
- عملکرد ضعیف با الگوهای داده بسیار تکراری
- اگر تعداد عناصر آرایه کم باشد، الگوریتم مرتب سازی سریع روش مناسبی نیست.
- عملکرد آن برای آرایه های مرتب شده یا تقریبا مرتب شده می‌تواند به طور قابل توجهی کاهش یابد.

https://www.techquintal.com/advantages-and-disadvantages-of-quick-sort/

پیاده‌سازی



from the scratch
Pseudo Code:

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Numpy.sort

```
np.sort(random_array, kind='quicksort')
```

توضیح الگوریتم

- Heap sort یا مرتب‌سازی هرمی یک الگوریتم مرتب سازی بر مبنای مقایسه است.
- یک هیپ دودویی (باینری) از آرایه ورودی ایجاد می‌کند.
- عنصر ماکزیمم (برای یک max-heap) یا مینیمم (برای یک min-heap) را از هیپ استخراج می‌کند. و در انتها آرایه مرتب شده قرار می‌دهد.
- و دوباره یک هیپ دیگر ایجاد می‌کند تا زمانی که آرایه مرتب شود.

ALGORITHM 230
MATRIX PERMUTATION
J. Boothroyd (Recd 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

```
procedure matrixperm(a,b,j,k,s,d,n,p); value n; real a,b;
  integer array s,d; integer j,k,n,p;
comment a procedure using Jensen's device which exchanges
  rows or columns of a matrix to achieve a rearrangement specified
  by the permutation vectors s,d[1:n]. Elements of s specify the
  original source locations while elements of d specify the desired
  destination locations. Normally a and b will be called as sub-
  scripted variables of the same array. The parameters j,k nom-
  inate the subscripts of the dimension affected by the permuta-
  tion, p is the Jensen parameter. As an example of the use of this
  procedure, suppose r,c[1:n] to contain the row and column sub-
  scripts of the successive matrix pivots used in a matrix inver-
  sion of an array a[1:n,1:n]; i.e. r[1], c[1] are the relative sub-
  scripts of the first pivot r[2], c[2] those of the second pivot and
  so on. The two calls
        matrixperm (a[j,p], a[k,p], j,k,r,c,n,p)
      and matrixperm (a[p,j], a[p,k], j,k,c,r,n,p)
  will perform the required rearrangement of rows and columns
    respectively;
begin integer array tag, loc[1:n]; integer i,t; real w;
comment set up initial vector tag number and address arrays;
  for i := 1 step 1 until n do tag[i] := loc[i] := i;
comment start permutation;
  for i := 1 step 1 until n do
    begin t := s[i]; j := loc[t]; k := d[i];
      if j≠k then begin for p := 1 step 1 until n do
                    begin w := a; a := b; b := w end;
                  tag[j] := tag[k]; tag[k] := t;
                  loc[t] := loc[tag[j]]; loc[tag[j]] := j
                end jk conditional
    end i loop
end matrixperm
```

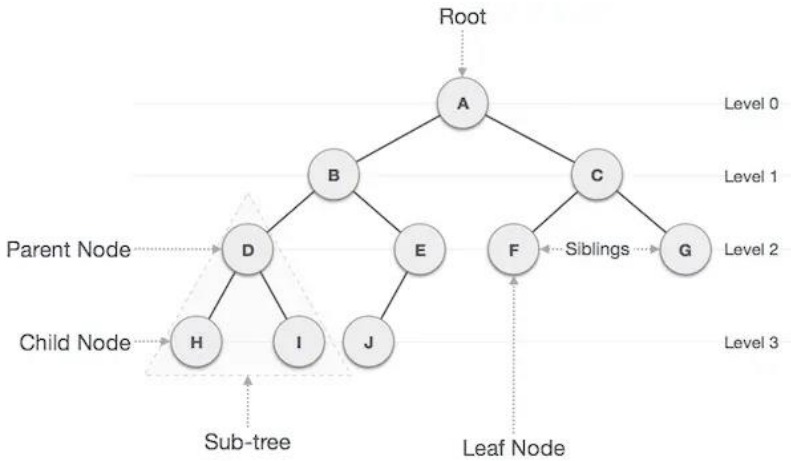
این الگوریتم ابتدا توسط J. W. J. Williams در سال 1964 پیشنهاد شد و سپس توسط R. W. Floyd در سال 1964 و J. W. J. Williams در سال 1965 بهبود یافت و منتشر شد.

https://dl.acm.org/doi/pdf/10.1145/512274.512284

نحوه کارکرد الگوریتم

- این الگوریتم یک max-heap (یا min-heap) از آرایه ورودی می‌سازد.

به این معنا که در گام اول یک درخت دودویی ایجاد می‌شود که به موجب آن مقدار گره والد بزرگتر از مقادیر دو گره فرزند است که به آن هرم بیشینه می‌گوییم (max-heap). اگر گره والد از فرزند خود کوچکتر باشد، هرم کمینه (min-heap) ایجاد می‌شود.



در درخت دودویی:

اندیس گره والد	i
اندیس فرزند چپ	$2i + 1$
اندیس فرزند راست	$2i + 2$

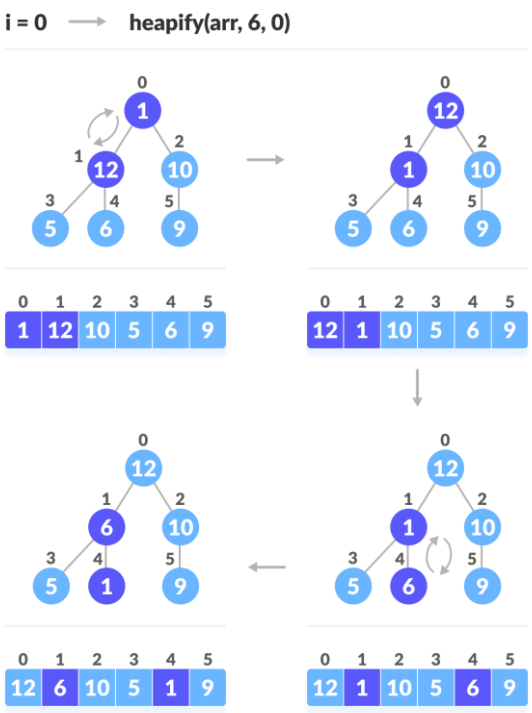
نحوه کارکرد الگوریتم

- با فرض ساخت هرم بیشینه در گام اول، در این حالت بزرگترین عنصر در گره (root) هرم قرار می‌گیرد.
- در گام بعدی بزرگترین مقدار از هرم حذف شده و در انتهای آرایه مرتب‌شده قرار می‌گیرد.
- پس از حذف بزرگترین مقدار، دوباره از مابقی اعداد، یک max-heap ساخته می‌شود تا دومین عدد بزرگ یافت شود و در ریشه قرار گیرد. دوباره این مقدار حذف و به مکان یکی قبل از انتهای آرایه منتقل می‌شود.
- گام‌های بالا تا زمانی که اندازه هرم بیش از یک باشد ادامه می‌یابد. و نهایتا آرایه مدنظر مرتب می‌شود.

پیچیدگی زمانی این الگوریتم $O(n \log n)$ است چرا که به میزان n بار عملیات حذف می‌بایست انجام شود و هر کدام از آن‌ها از $O(\log n)$ پیروی می‌کند.

https://fa.wikipedia.org/wiki/%D9%85%D8%B1%D8%AA%D8%A8%E2%80%8C%D8%B3%D8%A7%D8%B2%DB%8C_%D9%87%D8%B1%D9%85%DB%8C

مثالی از عملکرد الگوریتم

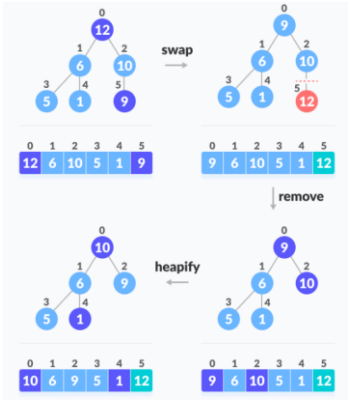


- در گام اول مطابق تصویر روبرو اولین heap ایجاد می‌شود.
- در این مثال عدد 12 که بزرگترین مقدار آرایه است مطابق فرآیند روبرو به root هرم منتقل می‌شود. (Heapify) (max-heap)
- در ادامه، عدد 12 از این هرم حذف می‌شود و به انتهای لیست منتقل می‌شود.
- سپس عدد 9 به گره منتقل می‌شود (آخرین عنصر از سمت راست هرم) و دوباره لازم است یک max-heap ایجاد شود که به موجب آن عدد 10 به root منتقل شود.
- سپس عدد 10 حذف و به انتهای آرایه یعنی قبل از عدد 12 منتقل می‌شود.
- این فرایند آنقدر تکرار می‌شود تا به آخرین عنصر آرایه برسیم. (مطابق تصویر صفحه بعد)

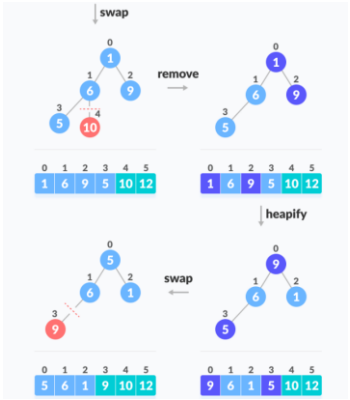
https://www.geeksforgeeks.org/quick-sort/

مثالی از عملکرد الگوریتم

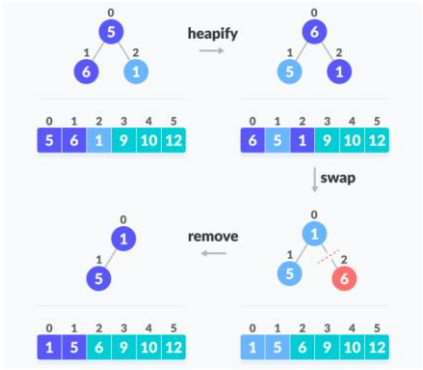
۱



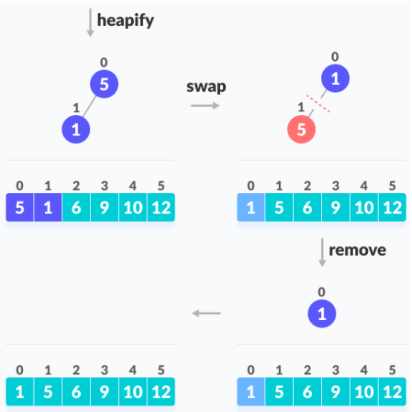
۲



۳



۴



مزایا و معایب

Pros

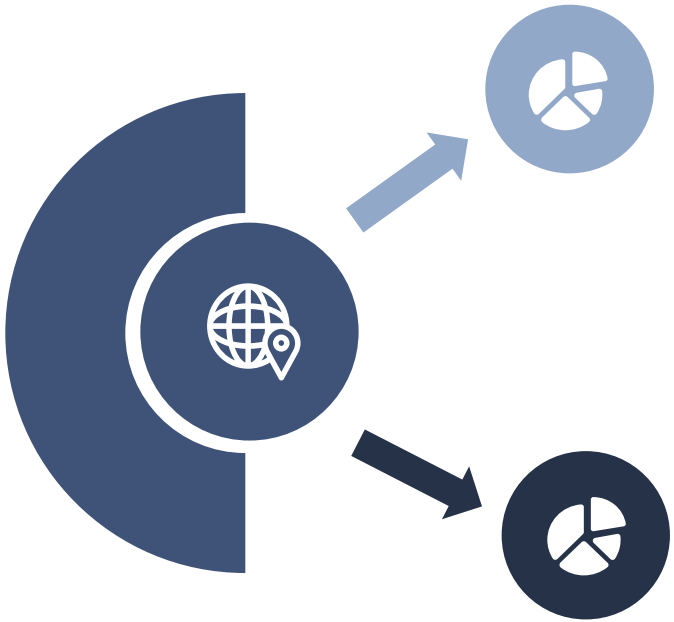
- In-place sorting: به این معنا که به حافظه اضافی برای مرتب سازی نیاز ندارد
- عملکرد خوب برای لیست‌های بزرگ
- تضمین پیچیدگی زمانی $O(n \log n)$ حتی در بدترین شرایط

Cons

- نیاز به حافظه اضافی برای انجام عملیات heapify
- عدم پایداری؛ ممکن است ترتیب نسبی عناصر مساوی را تغییر دهد.
- در عمل کندتر از quicksort و mergesort است.
- نامناسب برای دیتاست‌های کوچک به علت ساخت و نگهداری heap

https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-heap/
https://medium.com/dare-to-be-better/advantages-and-disadvantages-of-heap-data-structure-5343174c663c

پیاده‌سازی



from the scratch
Pseudo Code:

```
Heapsort(A) {
   BuildHeap(A)
   for i <- length(A) downto 2 {
      exchange A[1] <-> A[i]
      heapsize <- heapsize -1
      Heapify(A, 1)
}
```

```
BuildHeap(A) {
   heapsize <- length(A)
   for i <- floor( length/2 ) downto 1
      Heapify(A, i)
}
```

```
Heapify(A, i) {
   le <- left(i)
   ri <- right(i)
   if (le<=heapsize) and (A[le]>A[i])
      largest <- le
   else
      largest <- i
   if (ri<=heapsize) and (A[ri]>A[largest])
      largest <- ri
   if (largest != i) {
      exchange A[i] <-> A[largest]
      Heapify(A, largest)
   }
}
```

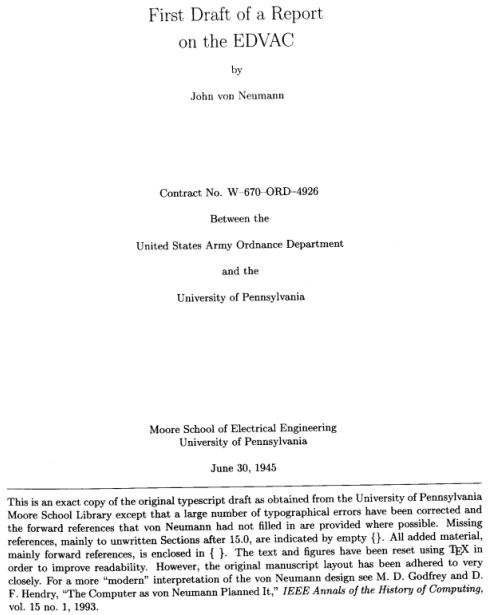
Numpy.sort

```
np.sort(random_array, kind='heapsort')
```

https://sites.cc.gatech.edu/classes/cslmth.trospaeh/llaf_98_3158

توضیح الگوریتم

- Merge Sort یا مرتب‌سازی ادغامی بر اساس استراتژی تقسیم و غلبه است.
- آرایه ورودی را به زیر آرایه‌هایی کوچکتر تقسیم می‌کند تا اینکه تنها شامل یک عضو شوند.
- هر کدام از زیرآرایه‌ها را به صورت بازگشتی مرتب می‌کند و سپس آن‌ها را با هم ادغام می‌کند.



First Draft of a Report
on the EDVAC

by

John von Neumann

Contract No. W–670–ORD–4926

Between the

United States Army Ordnance Department

and the

University of Pennsylvania

Moore School of Electrical Engineering
University of Pennsylvania

June 30, 1945

This is an exact copy of the original typescript draft as obtained from the University of Pennsylvania Moore School Library except that a large number of typographical errors have been corrected and the forward references that von Neumann had not filled in are provided where possible. Missing references, mainly to unwritten Sections after 15.0, are indicated by empty {}. All added material, mainly forward references, is enclosed in { }. The text and figures have been reset using TEX in order to improve readability. However, the original manuscript layout has been adhered to very closely. For a more "modern" interpretation of the von Neumann design see M. D. Godfrey and D. F. Hendry, "The Computer as von Neumann Planned It," *IEEE Annals of the History of Computing*, vol. 15 no. 1, 1993.

این الگوریتم توسط John von Neumann در سال 1945 هنگام کار بر روی کامپیوتر EDVAC ابداع شد.

https://web.mit.edu/STS.fdp.cavde/sFDP/www/035

نحوه کارکرد الگوریتم

مرحله تقسیم (Splitting Phase)

- در گام اول، آرایه ورودی به دو زیر لیست تقسیم می‌شود.
- فرآیند تقسیم برای هر کدام از زیرآرایه‌ها تا جایی ادامه می‌یابد که هر نهایتا هر کدام از آن‌ها تنها شامل یک عضو شوند تا برای انجام عملیات بازگشتی آماده شوند.

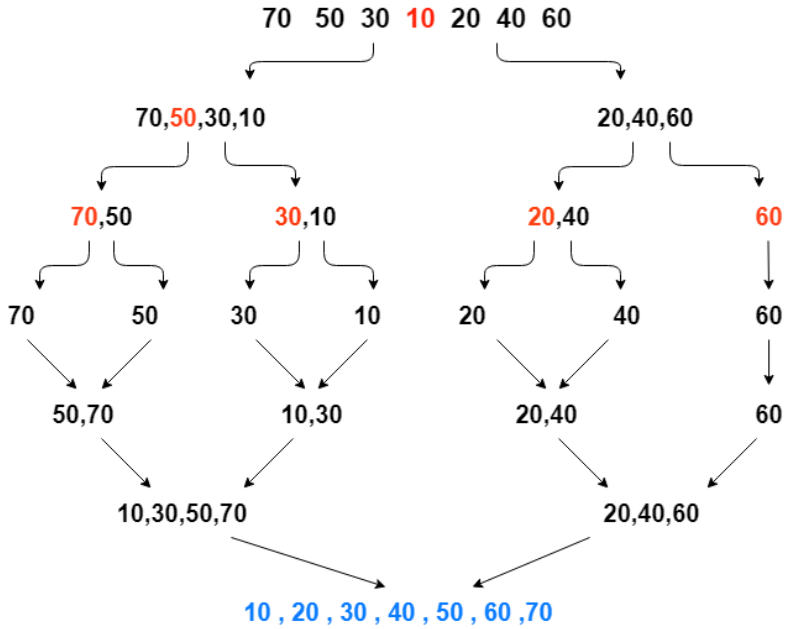
مرحله ادغام (Merging Phase)

- پس از اتمام مرحله تقسیم، الگوریتم شروع به ادغام زیرآرایه‌ها به صورت مرتب می‌کند.
- به موجب این الگوریتم، جفت‌های مجاور دو آرایه تک عنصری مقایسه می‌شوند و به صورت مرتب‌شده ادغام می‌شوند.
- فرآیند ادغام و مرتب سازی به صورت بازگشتی تا جایی ادامه می‌یابد که همه زیرآرایه‌ها به یک آرایه مرتب تبدیل شوند.

رابطه $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ بیانگر مرحله زمانی این الگوریتم است که $T(n)$ زمان لازم برای مرتب‌سازی یک آرایه به طول n است. به صورت کلی دارای پیچیدگی زمانی $O(n \log n)$ می‌باشد.

https://www.digitalocean.com/community/tutorials/merge-sort-algorithm-java-c-python

مثالی از عملکرد الگوریتم



- در گام اول، آرایه به طول n به $\frac{n}{2}$ تبدیل می‌شود.
- فرآیند تقسیم تا جایی ادامه می‌یابد که زیرآرایه‌های تک عضوی حاصل شوند.
- پس از اتمام فرآیند تقسیم، فرآیند ادغام زیرآرایه‌ها یکی یکی انجام می‌شود.
- نهایتا یک آرایه مرتب شده حاصل می‌شود.

https://www.digitalocean.com/community/tutorials/merge-sort-algorithm-java-c-python

مزایا و معایب

Pros

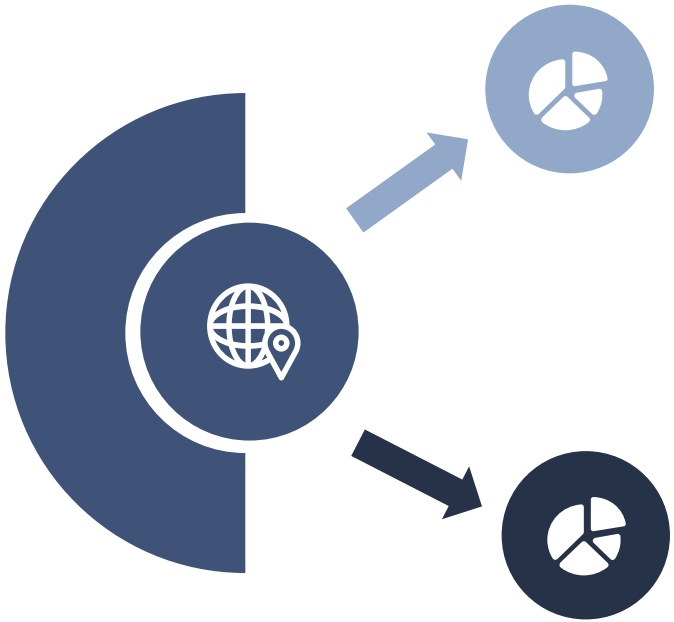
- عملکرد خوب برای لیست‌های بزرگ
- تضمین مرتبه زمانی $O(n \log n)$ حتی در بدترین شرایط
- برخلاف دو الگوریتم قبلی، پایدار است و ترتیب عناصر مساوی را تغییر نمی‌دهد.

Cons

- نیاز به حافظه اضافی برای انجام عملیات ادغام
- همانند الگوریتم‌های In-place sorting نظیر quicksort بهینه نیست.
- برای مرتب‌سازی آرایه‌های کوچک مناسب نیست.

https://ayush-arora.medium.com/sorting-algorithms-properties-pros-cons-comparisons-1b43599d5876

پیاده‌سازی



from the scratch
Pseudo Code:

```
def merge_sort(arr):
    if length of arr <= 1:
        return arr

    # Divide the array into two halves
    mid = length of arr // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort each half
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    # Merge the sorted halves
    return merge(left_half, right_half)
```

```
def merge(left, right):
    merged_array = []
    left_index = right_index = 0

    # Compare elements from left and right halves and merge
    while left_index < length of left and right_index < length of right:
        if left[left_index] < right[right_index]:
            merged_array.append(left[left_index])
            left_index += 1
        else:
            merged_array.append(right[right_index])
            right_index += 1

    # Add remaining elements from left half
    while left_index < length of left:
        merged_array.append(left[left_index])
        left_index += 1

    # Add remaining elements from right half
    while right_index < length of right:
        merged_array.append(right[right_index])
        right_index += 1

    return merged_array
```

Numpy.sort

```
np.sort(random_array, kind='mergesort')
```

https://cs.stackexchange.com/questions/tnairavni-pool-noitcnuf-egrem-strosegrem/155052

مقایسه الگوریتم های مرتب سازی

	speed	Time complexity worst case	Space complexity worst case	stable
Quick Sort	1	$O(n^2)$	$O(n)$	✗
Heap Sort	3	$O(n\ log\ n)$	$O(1)$	✗
Merge Sort	2	$O(n\ log\ n)$	$O(n)$	✓

- از آنجا که Heap Sort آرایه را به صورت In-place مرتب می‌کند لذا از لحاظ memory دارای مرتبه $O(1)$ است.
- اما Quick Sort در حالت میانگین و بدترین شرایط به ترتیب دارای مرتبه $O(log\ n)$ و $O(n)$ برای space complexity است چرا که فقط عملیات پارتیشن بندی را به صورت In-place انجام می‌دهد.
- نهایتا Merge Sort به دلیل نیاز به آرایه‌های موقت دارای مرتبه $O(n)$ است.

- بر حسب نوع و سایز آرایه ورودی هر کدام از این الگوریتم‌ها میتواند عملکرد بهتری داشته باشد.
- اگر در np.sort آرگمان `kind='stable'` انتخاب شود عملکردی شبیه به Merge Sort دارد.
- الگوریتم Quick Sort به عنوان پیش‌فرض برای np.sort در نظر گرفته شده‌است.

https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

از توجه شما عزیزان
سپاسگزارم