بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

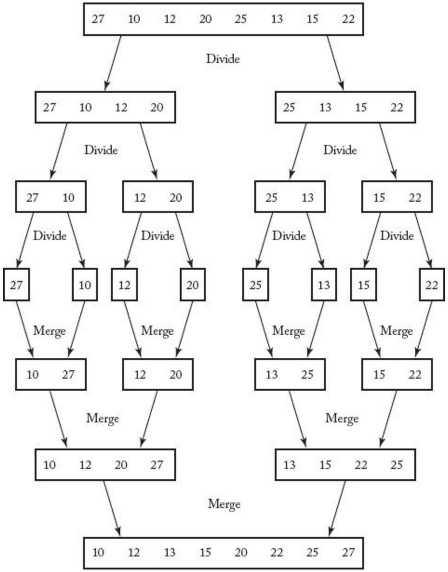
طراحی الگوریتم‌ها

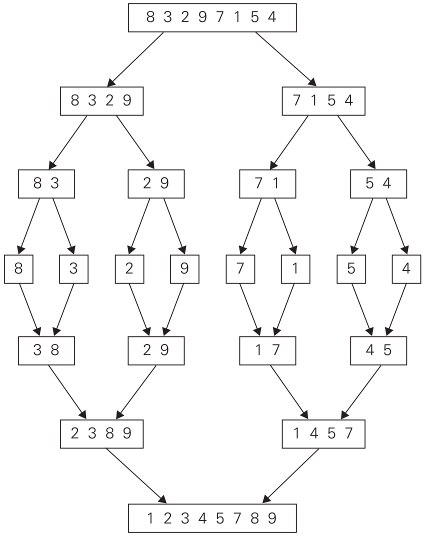
حسین فلسفین

Mergesort

To sort an array of 16 items, we can divide it into two subarrays, each of size 8, sort the two subarrays, and then merge them to produce the sorted array. In the same way, each subarray of size 8 can be divided into two subarrays of size 4, and these subarrays can be sorted and merged. Eventually, the size of the subarrays will become 1, and an array of size 1 is trivially sorted. This procedure is called "Mergesort."

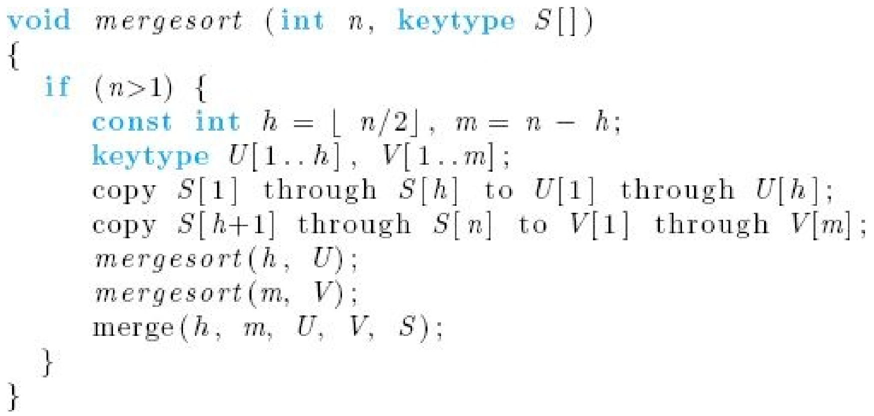
☞*Divide the array into two subarrays each with $\frac{n}{2}$ items.*
☞*Conquer (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.*
☞*Combine the solutions to the subarrays by merging them into a single sorted array.*


27 10 12 20 25 13 15 22
Divide
27 10 12 20
25 13 15 22
Divide
Divide
27 10
12 20
25 13
15 22
Divide
Divide
Divide
Divide
27
10
12
20
25
13
15
22
Merge
Merge
Merge
Merge
10 27
12 20
13 25
15 22
Merge
Merge
10 12 20 27
13 15 22 25
Merge
10 12 13 15 20 22 25 27


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

Mergesort

Problem: ***Sort*** n ***keys in nondecreasing sequence.***
Inputs: ***positive integer*** n***, array of keys*** S ***indexed from*** 1 ***to*** n***.***
Outputs: ***the array*** S ***containing the keys in nondecreasing order.***



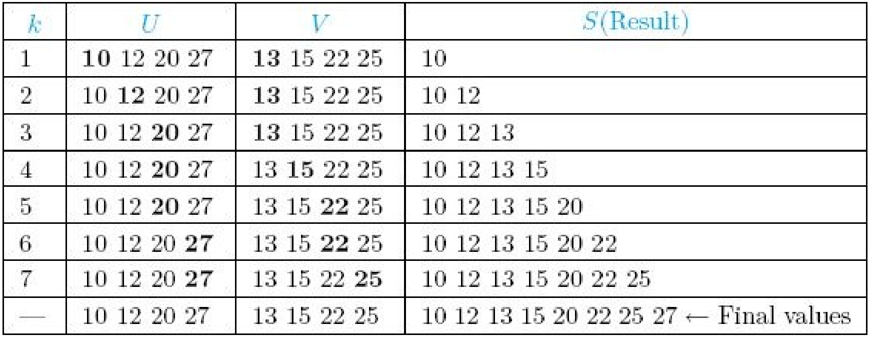
```
void mergesort (int n, keytype S[])
{
  if (n>1) {
     const int h = ⌊ n/2⌋, m = n − h;
     keytype U[1..h], V[1..m];
     copy S[1] through S[h] to U[1] through U[h];
     copy S[h+1] through S[n] to V[1] through V[m];
     mergesort(h, U);
     mergesort(m, V);
     merge(h, m, U, V, S);
  }
}
```

Merge

Problem: ***Merge two sorted arrays into one sorted array.***
Inputs: ***positive integers*** h ***and*** m***, array of sorted keys*** U ***indexed from*** 1 ***to*** h***, array of sorted keys*** V ***indexed from*** 1 ***to*** m***.***
Outputs: ***an array*** S ***indexed from*** 1 ***to*** $h+m$ ***containing the keys in*** U ***and*** V ***in a single sorted array.***

```
void merge (int h, int m, const keytype U[],
                          const keytype V[],
                                keytype S[])
{
  index i, j, k;

  i = 1; j = 1;  k = 1;
  while (i <= h && j <= m){
      if (U[i] < V[j]) {
          S[k] = U[i];
          i++;
      }
      else {
          S[k] = V[j];
          j++;
      }
      k++;
  }
  if (i>h)
      copy V[j] through V[m] to S[k] through S[h+m];
  else
      copy U[i] through U[h] to S[k] through S[h+m];
}
```



k	U	V	S(Result)
1	**10** 12 20 27	**13** 15 22 25	10
2	10 **12** 20 27	**13** 15 22 25	10 12
3	10 12 **20** 27	**13** 15 22 25	10 12 13
4	10 12 **20** 27	13 **15** 22 25	10 12 13 15
5	10 12 **20** 27	13 15 **22** 25	10 12 13 15 20
6	10 12 20 **27**	13 15 **22** 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 **25**	10 12 13 15 20 22 25
—	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27 $\leftarrow$ Final values

Worse-case time complexity of Merge

Basic operation: *the comparison of $U[i]$ with $V[j]$.*
Input size: *h and m, the number of items in each of the two input arrays.*
The worst case occurs when the loop is exited, because one of the indices—say, i—has reached its exit point $h+1$ whereas the other index j has reached m, 1 less than its exit point. For example, this can occur when the first $m-1$ items in V are placed first in S, followed by all h items in U, at which time the loop is exited because i equals $h+1$. Therefore,

$$W(h,m) = h + m - 1.$$

U	V	S
19, 24, 26, 30	**8**, 11, 17, 33	8
19, 24, 26, 30	8, **11**, 17, 33	8, 11
19, 24, 26, 30	8, 11, **17**, 33	8, 11, 17
19, 24, 26, 30	8, 11, 17, **33**	8, 11, 17, 19
19, **24**, 26, 30	8, 11, 17, **33**	8, 11, 17, 19, 24
19, 24, **26**, 30	8, 11, 17, **33**	8, 11, 17, 19, 24, 26
19, 24, 26, **30**	8, 11, 17, **33**	8, 11, 17, 19, 24, 26, 30
19, 24, 26, 30	8, 11, 17, 33	8, 11, 17, 19, 24, 26, 30, 33

Worst-case time complexity of Mergesort

***Basic operation:* the comparison that takes place in merge.**

***Input size:* n, the number of items in the array S.**

$$W(n) = \underbrace{W(h)}_{\text{Time to sort } U} + \underbrace{W(m)}_{\text{Time to sort } V} + \underbrace{h+m-1}_{\text{Time to merge}}$$

Let n be a power of 2. In this case,

$$h = \lfloor n/2 \rfloor = \frac{n}{2}$$

$$m = n - h = n - \frac{n}{2} = \frac{n}{2}$$

$$h + m = \frac{n}{2} + \frac{n}{2} = n.$$

Our expression for $W(n)$ becomes

$$W(n) = W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + n - 1$$

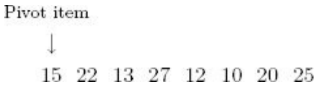
$$= 2W\left(\frac{n}{2}\right) + n - 1.$$

$$\boxed{\begin{array}{ll} W(n) = 2W\left(\frac{n}{2}\right) + n - 1 & \text{for } n > 1,\ n \text{ a power of } 2 \\ W(1) = 0 & \end{array}}$$

Master theorem: $W(n) \in \Theta(n \log n)$

Quicksort

Quicksort is similar to Mergesort in that the sort is accomplished by dividing the array into two partitions and then sorting each partition recursively. In Quicksort, however, the array is partitioned by placing all items smaller than some pivot item before that item and all items larger than the pivot item after it. The pivot item can be any item, and for convenience we will simply make it the first one.



$$\underbrace{10\quad 13\quad 12}_{\text{All smaller}}\quad \overset{\overset{\text{Pivot item}}{\downarrow}}{15}\quad \underbrace{22\quad 27\quad 20\quad 25}_{\text{All larger}}$$

$$\underbrace{10\quad 12\quad 13}_{\text{Sorted}}\quad \overset{\overset{\text{Pivot item}}{\downarrow}}{15}\quad \underbrace{20\quad 22\quad 25\quad 27}_{\text{Sorted}}$$

After the partitioning, the order of the items in the subarrays is unspecified and is a result of how the partitioning is implemented. We have ordered them according to how the partitioning routine, which will be presented shortly, would place them. The important thing is that all items smaller than the pivot item are to the left of it, and all items larger are to the right of it. Quicksort is then called recursively to sort each of the two subarrays. They are partitioned, and this procedure is continued until an array with one item is reached. Such an array is trivially sorted.

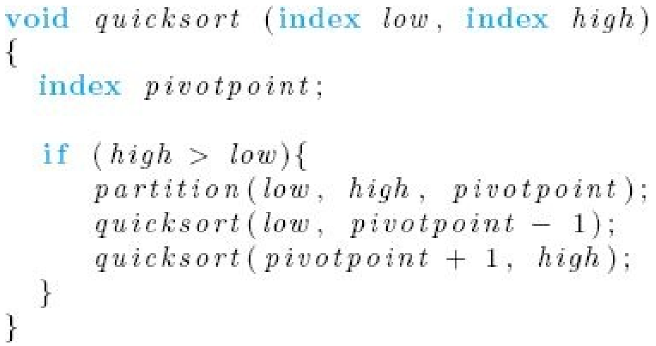
لذا در فرایند ***Quicksort***، ***combine*** نداریم.

If the algorithm were implemented by defining S globally and n was the number of items in S, the top-level call to quicksort would be $quicksort(1, n);$

Problem: Sort n keys in nondecreasing order.

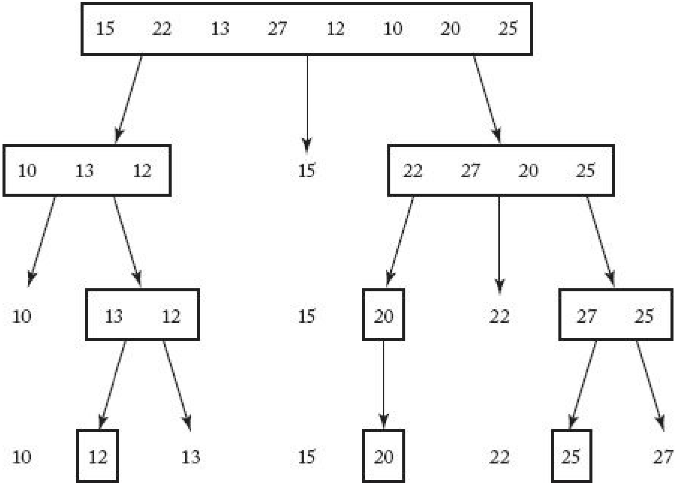
Inputs: positive integer n, array of keys S indexed from 1 to n.

Outputs: the array S containing the keys in nondecreasing order.



```
void quicksort (index low, index high)
{
  index pivotpoint;

  if (high > low){
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint − 1);
     quicksort(pivotpoint + 1, high);
  }
}
```


15 22 13 27 12 10 20 25
10 13 12
15
22 27 20 25
10
13 12
15
20
22
27 25
10
12
13
15
20
22
25
27

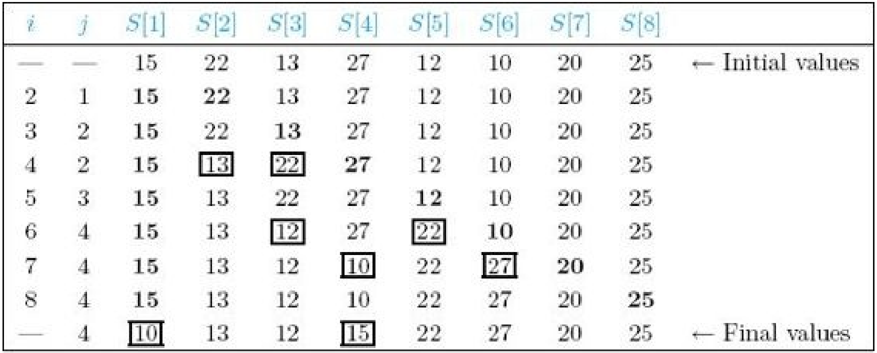
***Problem:* Partition the array S for Quicksort.**

***Inputs:* two indices, low and $high$, and the subarray of S indexed from low to $high$.**

***Outputs:* $pivotpoint$, the pivot point for the subarray indexed from low to $high$.**

```
void partition (index low, index high,
                index& pivotpoint)
{
   index i, j;
   keytype pivotitem;

   pivotitem = S[low];                    // Choose first item for
   j = low;                               // pivotitem.
   for (i = low + 1; i <= high; i++)
      if (S[i] < pivotitem){
         j++;
         exchange S[i] and S[j];
      }
   pivotpoint = j;
   exchange S[low] and S[pivotpoint];  // Put pivotitem at pivotpoint.
}
```

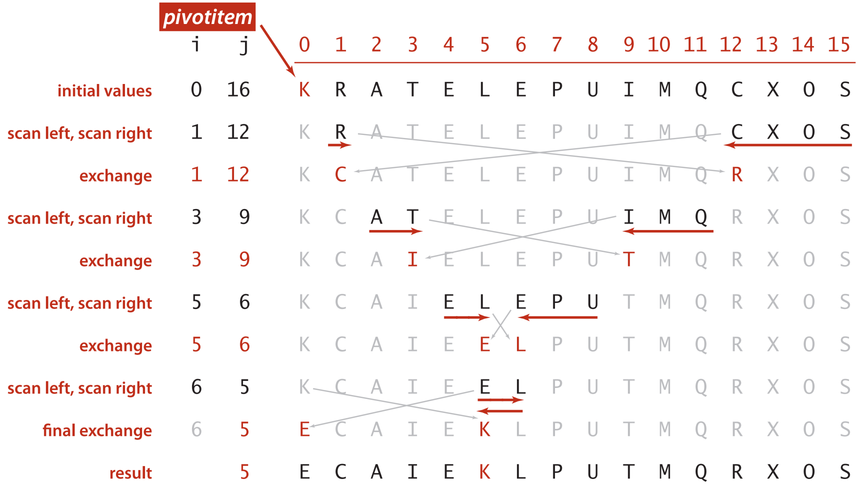


i	j	$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$	$S[6]$	$S[7]$	$S[8]$	
—	—	15	22	13	27	12	10	20	25	← Initial values
2	1	**15**	**22**	13	27	12	10	20	25	
3	2	**15**	22	**13**	27	12	10	20	25	
4	2	**15**	$\boxed{13}$	$\boxed{22}$	**27**	12	10	20	25	
5	3	**15**	13	22	27	**12**	10	20	25	
6	4	**15**	13	$\boxed{12}$	27	$\boxed{22}$	**10**	20	25	
7	4	**15**	13	12	$\boxed{10}$	22	$\boxed{27}$	**20**	25	
8	4	**15**	13	12	10	22	27	20	**25**	
—	4	$\boxed{10}$	13	12	$\boxed{15}$	22	27	20	25	← Final values

یک الگوریتم **partition** متفاوت که برای آن هم داریم $T(n) \in \Theta(n)$

First, we choose $S[low]$ to be the partitioning item. Next, we scan from the left end of the array until we find an entry greater than (or equal to) the partitioning item, and we scan from the right end of the array until we find an entry less than (or equal to) the partitioning item. The two items that stopped the scans are out of place in the final partitioned array, so we exchange them. Continuing in this way, we ensure that no array entries to the left of the left index i are greater than the partitioning item, and no array entries to the right of the right index j are less than the partitioning item. When the scan indices cross, all that we need to do to complete the partitioning process is to exchange the partitioning item $S[low]$ with the rightmost entry of the left subarray ($S[j]$) and return its index j.

یک الگوریتم *partition* متفاوت که برای آن هم داریم $T(n) \in \Theta(n)$



	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Partitioning trace (array contents before and after each exchange)

Every-case time complexity of $partition$

***Basic operation:* the comparison of $S[i]$ with $pivotitem$.**

***Input size:* $n = high - low + 1$, the number of items in the subarray.**

Because every item except the first is compared,

$$T(n) = n - 1.$$

Quicksort does not have an every-case complexity. We will do worst-case and average-case analyses.

Worst-case time complexity of $quicksort$

Basic operation: ***the comparison of $S[i]$ with $pivotitem$ in partition.***
Input size: ***n, the number of items in the array S.***

Oddly enough, it turns out that the worst case occurs if the array is already sorted in nondecreasing order. If the array is already sorted in nondecreasing order, no items are less than the first item in the array, which is the pivot item. Therefore, when partition is called at the top level, no items are placed to the left of the pivot item, and the value of pivotpoint assigned by partition is 1. Similarly, in each recursive call, $pivotpoint$ receives the value of low. Therefore, the array is repeatedly partitioned into an empty subarray on the left and a subarray with one less item on the right.

For the class of instances that are already sorted in nondecreasing order, we have

$$T(n) = \underbrace{T(0)}_{\substack{\text{Time to sort}\\ \text{left subarray}}} + \underbrace{T(n-1)}_{\substack{\text{Time to sort}\\ \text{right subarray}}} + \underbrace{n-1}_{\substack{\text{Time to}\\ \text{partition}}}$$

We are using the notation $T(n)$ because we are presently determining the every-case complexity for the class of instances that are already sorted in nondecreasing order.

$$\boxed{\begin{array}{ll} T(n) = T(n-1) + n - 1 & \text{for } n > 0 \\ T(0) = 0. & \end{array}}$$

$$T(n) = \frac{n(n-1)}{2}$$

با بهره‌گیری از استقرا می‌توان رسماً ثابت کرد که $W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$.

Average-case time complexity of $quicksort$

***Basic operation:* the comparison of $S[i]$ with $pivotitem$ in partition**

***Input size:* n, the number of items in the array S.**

$$A(n)=\sum_{p=1}^{n}\overset{\substack{\text{Probability}\\ pivotpoint\text{ is }p\\ \downarrow}}{\frac{1}{n}}\underbrace{[A(p-1)+A(n-p)]}_{\substack{\text{Average time to}\\ \text{sort subarrays when}\\ pivotpoint\text{ is }p}}+\underbrace{n-1}_{\substack{\text{Time to}\\ \text{partition}}}$$

از طرفی داریم:

$$\sum_{p=1}^{n}\left[A\left(p-1\right)+A\left(n-p\right)\right]=2\sum_{p=1}^{n}A\left(p-1\right).$$

Why? Because

$$\sum_{p=1}^{n}(A(p-1)+A(n-p)) = \sum_{p=1}^{n} A(p-1) + \sum_{p=1}^{n} A(n-p) =$$

$$(A(0)+A(1)+A(2)+\cdots+A(n-1))+$$

$$(A(n-1)+A(n-2)+A(n-3)+\cdots+A(0)) = 2\sum_{p=1}^{n} A(p-1).$$

Therefore,

$$A(n) = \frac{2}{n}\sum_{p=1}^{n} A(p-1) + n - 1.$$

دو طرف این رابطه را در n ضرب کنید:

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1).$$

اگر در رابطهٔ فوق به‌جای n قرار دهیم $n-1$ داریم:

$$(n-1)A(n-1)=2\sum_{p=1}^{n-1}A(p-1)+(n-1)(n-2).$$

حال اگر رابطهٔ آخر را از رابطهٔ یکی به آخر مانده کم کنیم داریم:

$$nA(n)-(n-1)A(n-1)=2A(n-1)+2(n-1),$$

دو طرف رابطهٔ نهایی را تقسیم بر $n(n+1)$ می‌کنیم:

$$\frac{A(n)}{n+1}=\frac{A(n-1)}{n}+\frac{2(n-1)}{n(n+1)}.$$

اگر قرار دهیم

$$a_n = \frac{A(n)}{n+1},$$

آنگاه خواهیم داشت:

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \qquad \text{for } n > 0$$
$$a_0 = 0.$$

It can be shown that $a_n \approx 2\ln n$ ***which implies that***

$$A(n) \approx (n+1)\,2\ln n = (n+1)\,2(\ln 2)(\lg n)$$
$$\approx 1.38\,(n+1)\lg n \in \Theta(n \lg n).$$