Quicksort

❖ Quicksort

Previous sorts were all $O(n^k)$ (where $k > 1$).

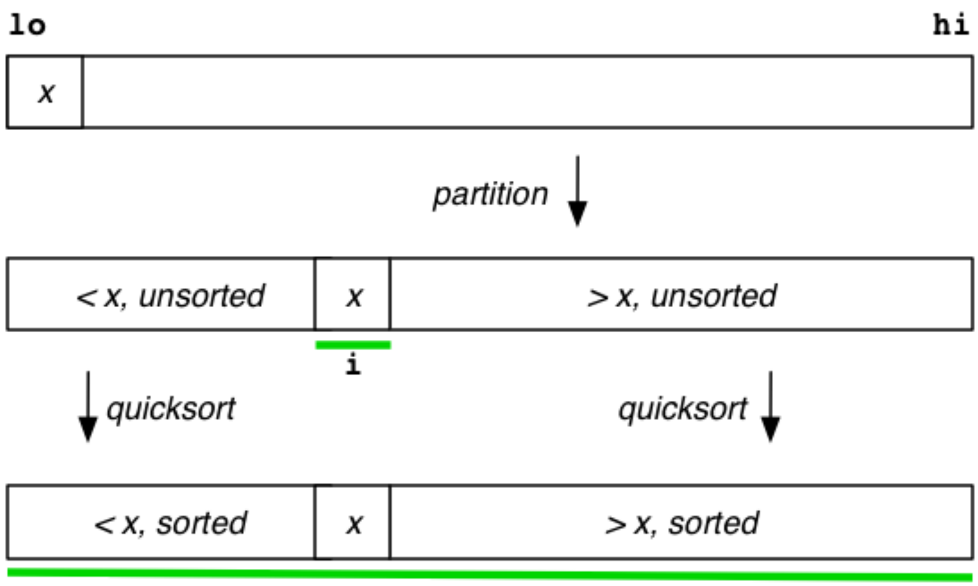
We can do better ...

Quicksort: basic idea

- choose an item to be a "pivot" 枢轴
- re-arrange (partition 分区) the array so that
  left < p < right
  - all elements to left of pivot are smaller than pivot
  - all elements to right of pivot are greater than pivot
- (recursively) sort each of the partitions

❖ ... Quicksort

Phases of quicksort:



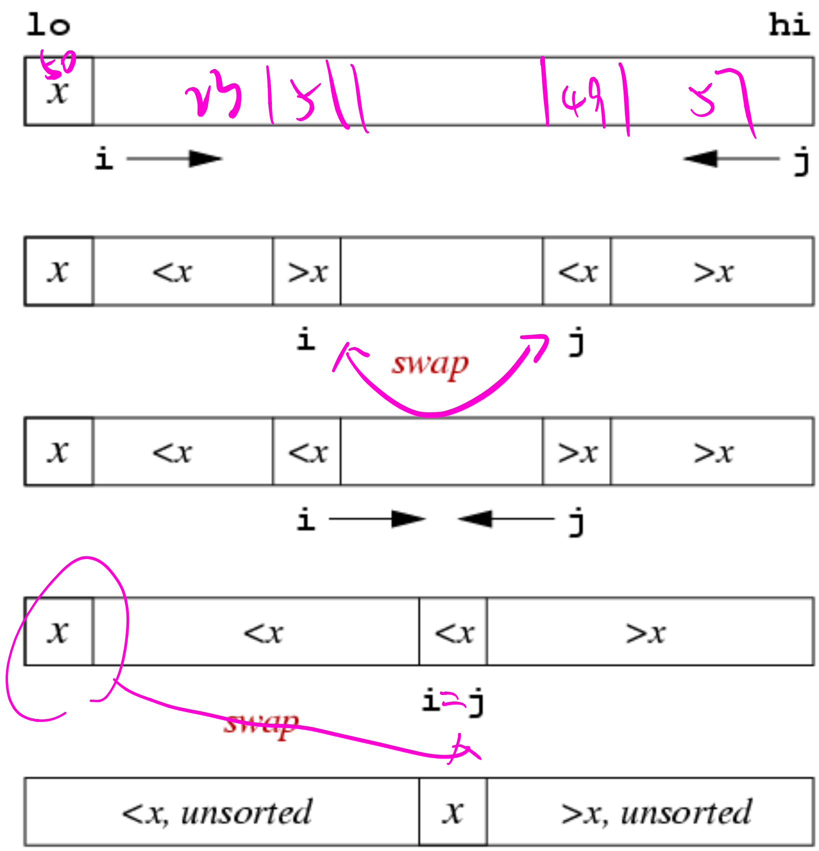
❖ Quicksort Implementation

Elegant recursive solution ...

```
void quicksort(Item a[], int lo, int hi)
{
    int i; // index of pivot
    if (hi <= lo) return;
    i = partition(a, lo, hi);
    quicksort(a, lo, i-1);
    quicksort(a, i+1, hi);
}
```

❖ ... Quicksort Implementation

Partitioning phase:



❖ ... Quicksort Implementation

Partition implementation:

```
int partition(Item a[], int lo, int hi)
{
   Item v = a[lo];   // pivot
   int  i = lo+1, j = hi;
   for (;;) {
      while (less(a[i],v) && i < j) i++;
      while (less(v,a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
   }
   j = less(a[i],v) ? i : i-1;
   swap(a,lo,j);
   return j;
}
```

first

a[hi]

a[i] > v   v > a[j]

if (a[i] < v) {
j = i
} else { (a[i] > v)
j = i-1 }

❖ Quicksort Performance

Best case: $O(nlogn)$ comparisons

- choice of pivot gives middle two equal-sized partitions
- same happens at every recursive level
- each "level" requires approx n comparisons
- halving at each level ⇒ log_2n levels

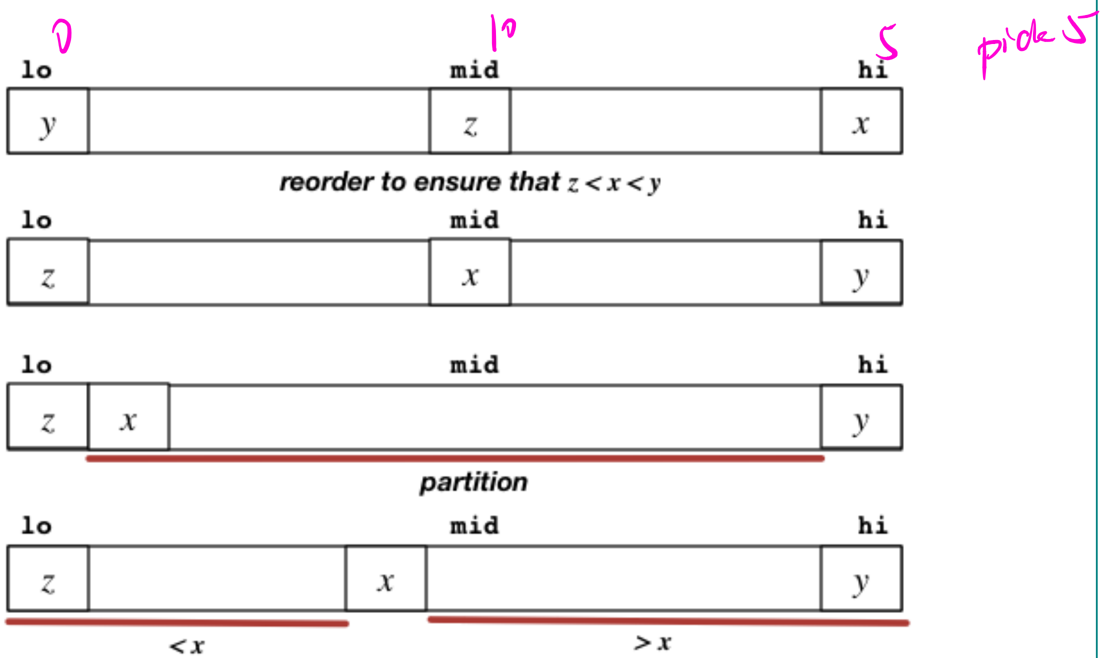
Worst case: $O(n^2)$ comparisons

- always choose lowest/highest value for pivot
- partitions are size 1 and $n-1$
- each "level" requires approx n comparisons
- partitioning to 1 and $n-1$ ⇒ n levels

❖ Quicksort Improvements

Choice of pivot can have significant effect:

- always choosing largest/smallest ⇒ worst case
- try to find "intermediate" value by median-of-three



❖ ... Quicksort Improvements

Median-of-three partitioning:

```
void medianOfThree(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   if (less(a[hi],a[mid])) swap(a, mid, hi);
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   // now, we have a[lo] < a[mid] < a[hi]
   // swap a[mid] to a[lo+1] to use as pivot
   swap(a, mid, lo+1);
}
void quicksort(Item a[], int lo, int hi)
{
   if (hi <= lo) return;
   medianOfThree(a, lo, hi);
   int i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

hi 一定比 middle 大

不管他了.

middle

❖ ... Quicksort Improvements

Another source of inefficiency:

- pushing recursion down to very small partitions
- overhead in recursive function calls
- little benefit from partitioning when size < 5

Solution: handle small partitions differently

- switch to insertion sort on small partitions, or
- don't sort yet; use post-quicksort insertion sort

❖ ... Quicksort Improvements

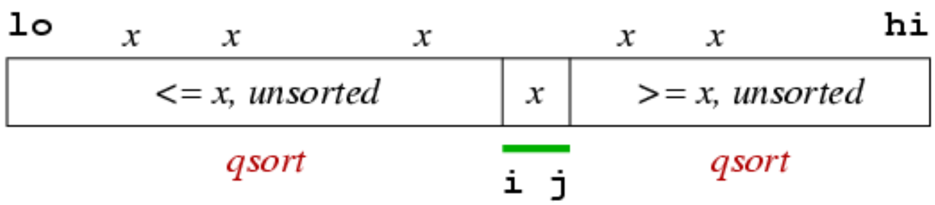
Quicksort with thresholding ...

```
void quicksort(Item a[], int lo, int hi)
{                                5
   if (hi-lo < Threshhold) {
      insertionSort(a, lo, hi);
      return;
   }
   medianOfThree(a, lo, hi);
   int i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

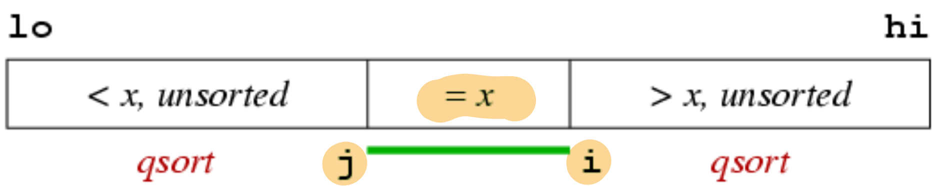
❖ ... Quicksort Improvements

If the array contains many duplicate keys

- standard partitioning does not exploit this

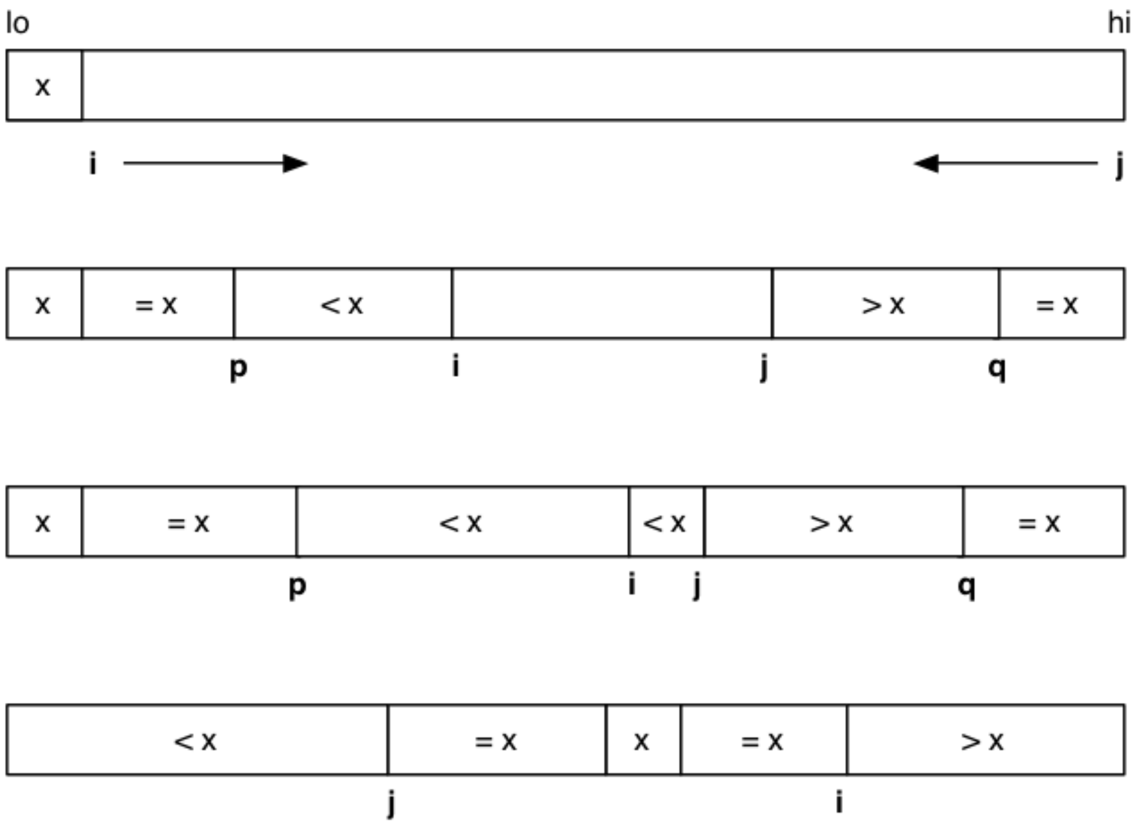


- can improve performance via three-way partitioning



❖ ... Quicksort Improvements

Bentley/McIlroy approach to three-way partition:



❖ Non-recursive Quicksort

Quicksort can be implemented using an explicit stack:

```
void quicksortStack (Item a[], int lo, int hi)
{
   Stack s = newStack();
   StackPush(s,hi); StackPush(s,lo);
   while (!StackEmpty(s)) {
      lo = StackPop(s); 抓走
      hi = StackPop(s);
      if (hi > lo) {
         int i = partition (a,lo,hi);
         StackPush(s,hi); StackPush(s,i+1);
         StackPush(s,i-1); StackPush(s,lo);
      }
   }
}
```

Mergesort

❖ Mergesort

Mergesort: basic idea

- split the array into two equal-sized paritions
- (recursively) sort each of the partitions
- merge the two partitions into a new sorted array O(n)
- copy back to original array

Merging: basic idea

- copy elements from the inputs one at a time
- give preference to the smaller of the two O(n logn)
- when one exhausted, copy the rest of the other

Merging.

which is smaller

Sorted : 4 12 17 23 59 92

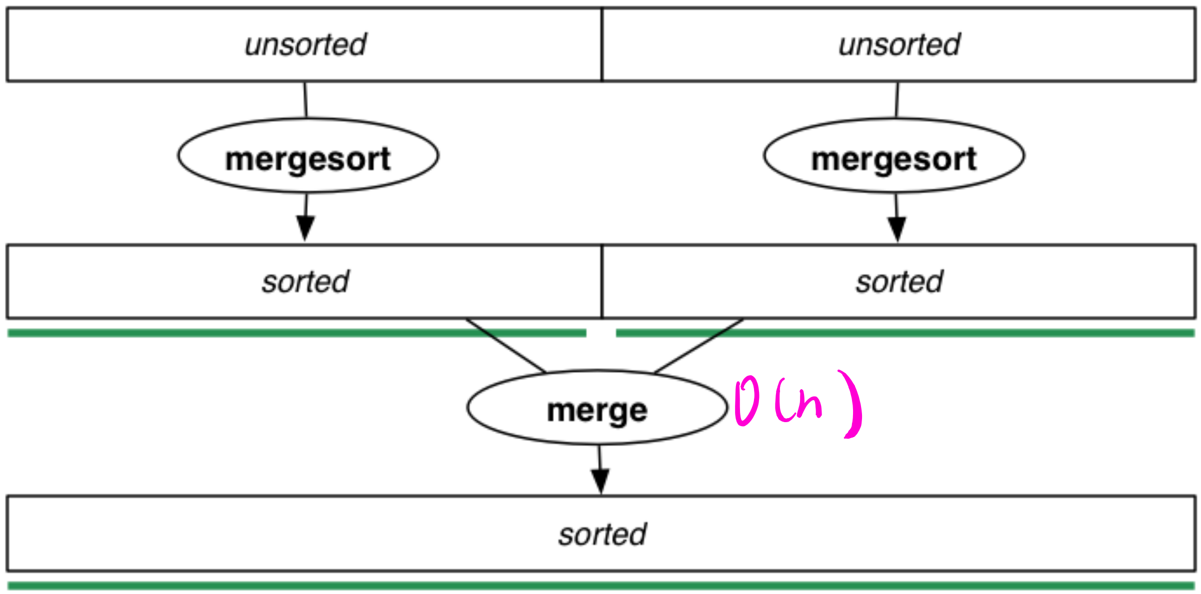
1 3 12 20 51

sorted :

Output : 1 3 4 12 12 17 20 , 51 , 59 92

❖ ... Mergesort

Phases of mergesort



❖ Mergesort Implementation

Mergesort function:

```
void mergesort(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2; // mid point
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
   merge(a, lo, mid, hi);
}
```

Example of use (**`typedef int Item`**):

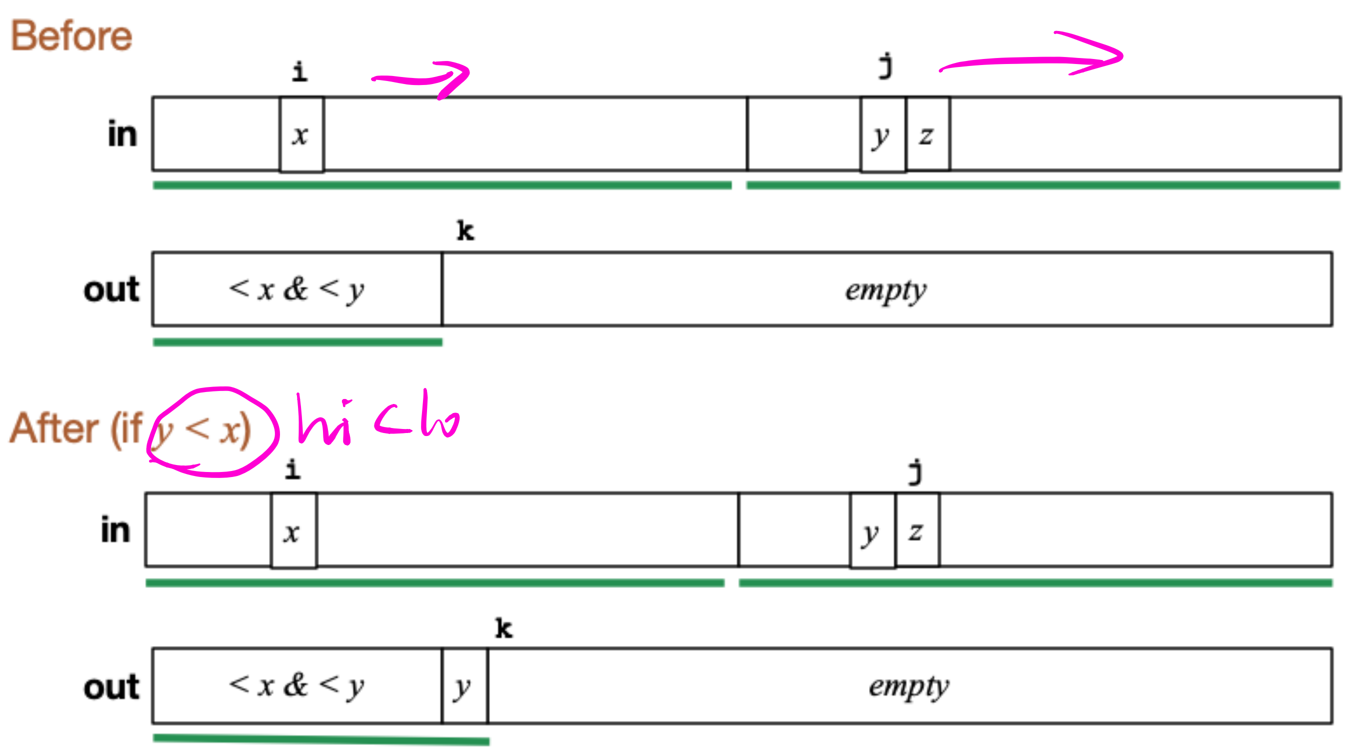
0 9

```
int nums[10] = {32,45,17,22,94,78,64,25,55,42};
mergesort(nums, 0, 9);
```

first last

❖ ... Mergesort Implementation

One step in the merging process:



❖ ... Mergesort Implementation

Implementation of merge:

1 2

```
void merge(Item a[], int lo, int mid, int hi)
{
   int  i, j, k, nitems = hi-lo+1;
   Item *tmp = malloc(nitems*sizeof(Item));

   i = lo; j = mid+1; k = 0;
   // scan both segments, copying to tmp
   while (i <= mid && j <= hi) { lo->mid  mid+1->hi
     if (less(a[i],a[j]))
        tmp[k++] = a[i++]; 谁小谁加入
     else
        tmp[k++] = a[j++];
   }
   // copy items from unfinished segment
   while (i <= mid) tmp[k++] = a[i++];
   while (j <= hi) tmp[k++] = a[j++];

   //copy tmp back to main array
   for (i = lo, k = 0; i <= hi; i++, k++)
      a[i] = tmp[k];
   free(tmp);
}
```

❖ Mergesort Performance

Best case: *O(NlogN)* comparisons

- split array into equal-sized partitions
- same happens at every recursive level
- each "level" requires $\le N$ comparisons
- halving at each level ⇒ $log_2 N$ levels

Worst case: *O(NlogN)* comparisons

- partitions are exactly interleaved
- need to compare all the way to end of partitions

Disadvantage over quicksort: need extra storage *O(N)*

❖ Bottom-up Mergesort

Non-recursive mergesort does not require a stack

- partition boundaries can be computed iteratively

Bottom-up mergesort:

- on each pass, array contains sorted runs of length m
- at start, treat as N sorted runs of length 1
- 1st pass merges adjacent elements into runs of length 2
- 2nd pass merges adjacent 2-runs into runs of length 4
- continue until a single sorted run of length N

This approach can be used for "in-place" mergesort.

❖ ... Bottom-up Mergesort

	[0]	[1]	[2]					.....								[15]
Original	A	S	O	R	T	I	N	G	E	X	E	M	P	L	A	R
After 1st pass *sorted slices of length 2*	A	S	O	R	I	T	G	N	E	X	E	M	L	P	A	R
After 2nd pass *sorted slices of length 4*	A	O	R	S	G	I	N	T	E	E	M	X	A	L	P	R
After 3rd pass *sorted slices of length 8*	A	G	I	N	O	R	S	T	A	E	E	L	M	P	R	X
After 4th pass *sorted slice of length 16*	A	A	E	E	G	I	L	M	N	O	P	R	R	S	T	X

merge

❖ ... Bottom-up Mergesort

Bottom-up mergesort for arrays:

```
#define min(A,B) ((A < B) ? A : B)

void mergesort(Item a[], int lo, int hi)
{
    int m;      // m = length of runs
    int len;    // end of 2nd run
    Item c[];  // array to merge into
    for (m = 1; m <= lo-hi; m = 2*m) {
        for (int i = lo; i <= hi-m; i += 2*m) {
            len = min(m, hi-(i+m)+1);
            merge(&a[i], m, &a[i+m], len, c);
            copy(&a[i], c, m+len);
        }
    }
}
```

number of elements

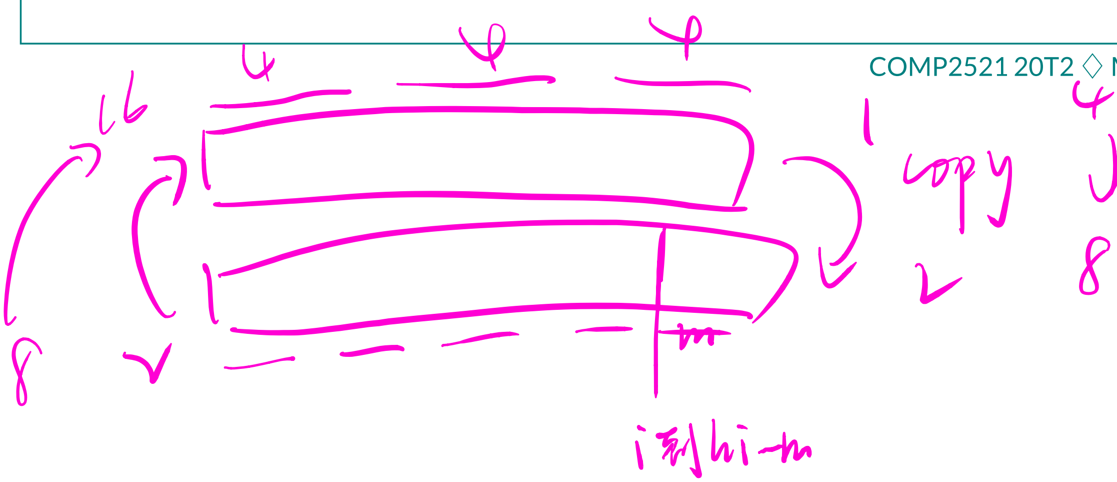
1 have

start

4 elem

end

should be m



❖ ... Bottom-up Mergesort

```
                              merge
// merge arrays a[] and b[] into c[]
// aN = size of a[], bN = size of b[]
void merge(Item a[], int aN, Item b[], int bN, Item c[])
                            m
{
   int i; // index into a[]
   int j; // index into b[]
   int k; // index into c[]  2m
   for (i = j = k = 0; k < aN+bN; k++) {
      if (i == aN) end of a[]
         c[k] = b[j++];
      else if (j == bN)
         c[k] = a[i++];  <
      else if (less(a[i],b[j]))
         c[k] = a[i++];
      else                a[i] > b[j]
         c[k] = b[j++];
   }
}
```

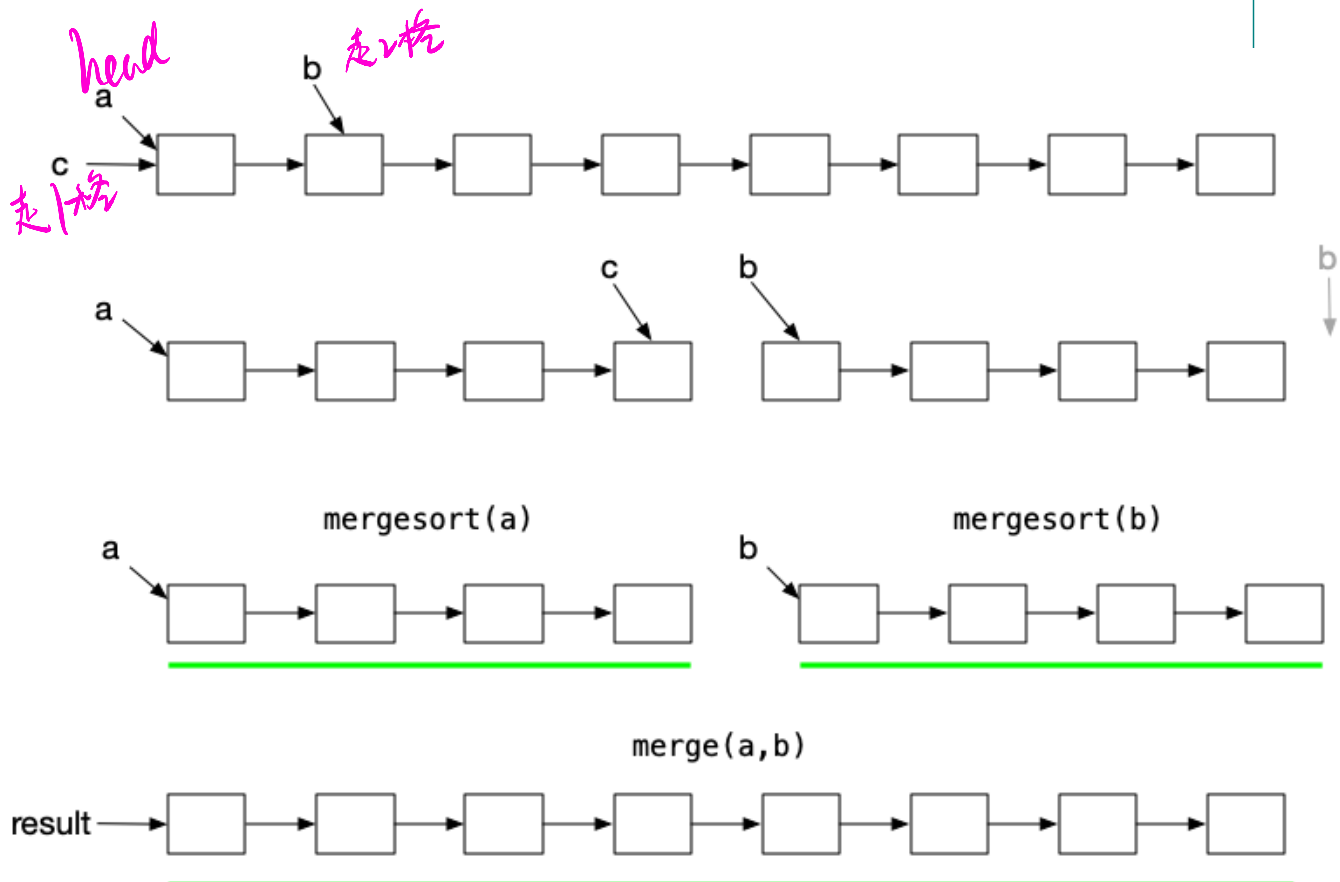
❖ Mergesort and Linked Lists

Merging linked lists is relatively straightforward:

```
List merge(List a, List b)
{
   List new = newList();
   while (a != NULL && b != NULL) {
      if (less(a->item, b->item))
         { new = ListAppend(new, a->item); a = a->next; }
      else a>b
         { new = ListAppend(new, b->item); b = b->next; }
   }
   while (a != NULL)
      { new = ListAppend(new, a->item); a = a->next; }
   while (b != NULL)
      { new = ListAppend(new, b->item); b = b->next; }
   return new;
}
```

❖ ... Mergesort and Linked Lists

Mergesort method using linked lists



❖ ... Mergesort and Linked Lists

Recursive linked list mergesort, built with list merge:

```
List mergesort(List c)
{
   List a, b;
   if (c == NULL || c->next == NULL) return c;
   a = c;  b = c->next;
   while (b != NULL && b->next != NULL)
      { c = c->next; b = b->next->next; }
   b = c->next; c->next = NULL;   // split list
   return merge(mergesort(a), mergesort(b));
}
```