Mergesort

❖ Mergesort

Mergesort: basic idea

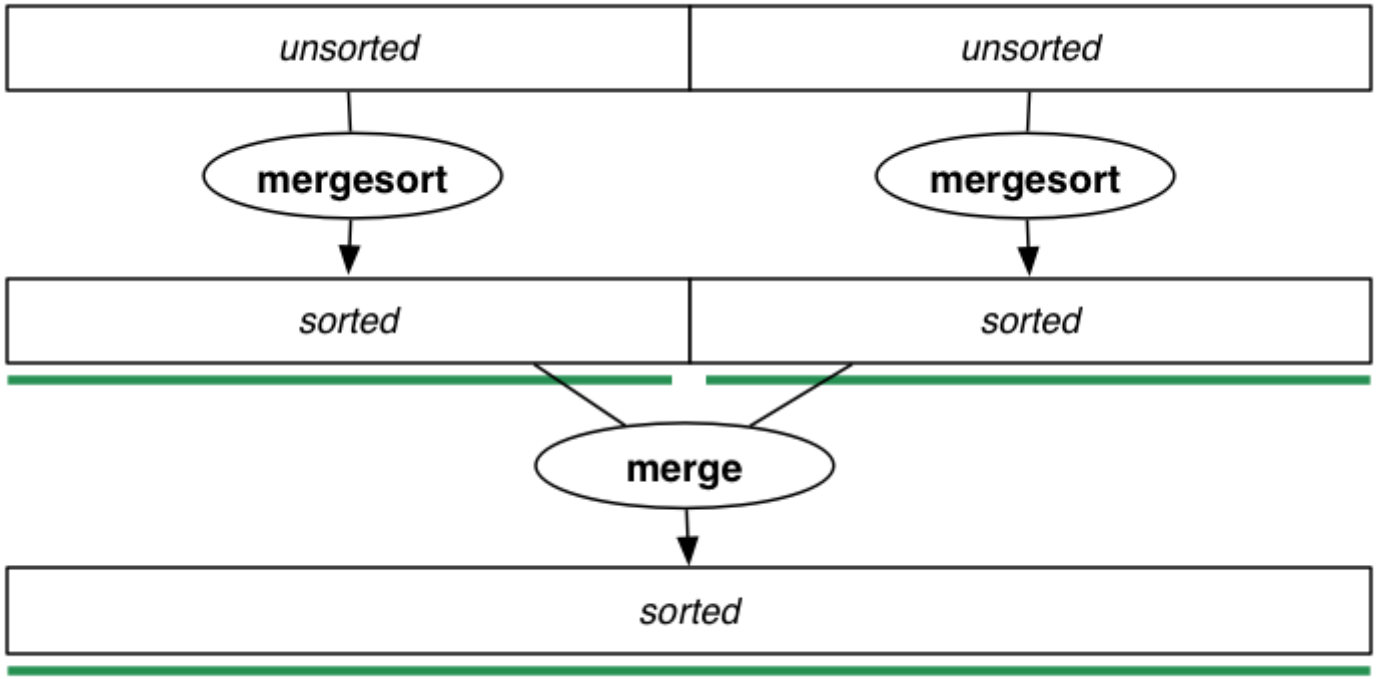
- split the array into two equal-sized paritions
- (recursively) sort each of the partitions
- merge the two partitions into a new sorted array
- copy back to original array

Merging: basic idea

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

❖ ... Mergesort

Phases of mergesort



❖ Mergesort Implementation

Mergesort function:

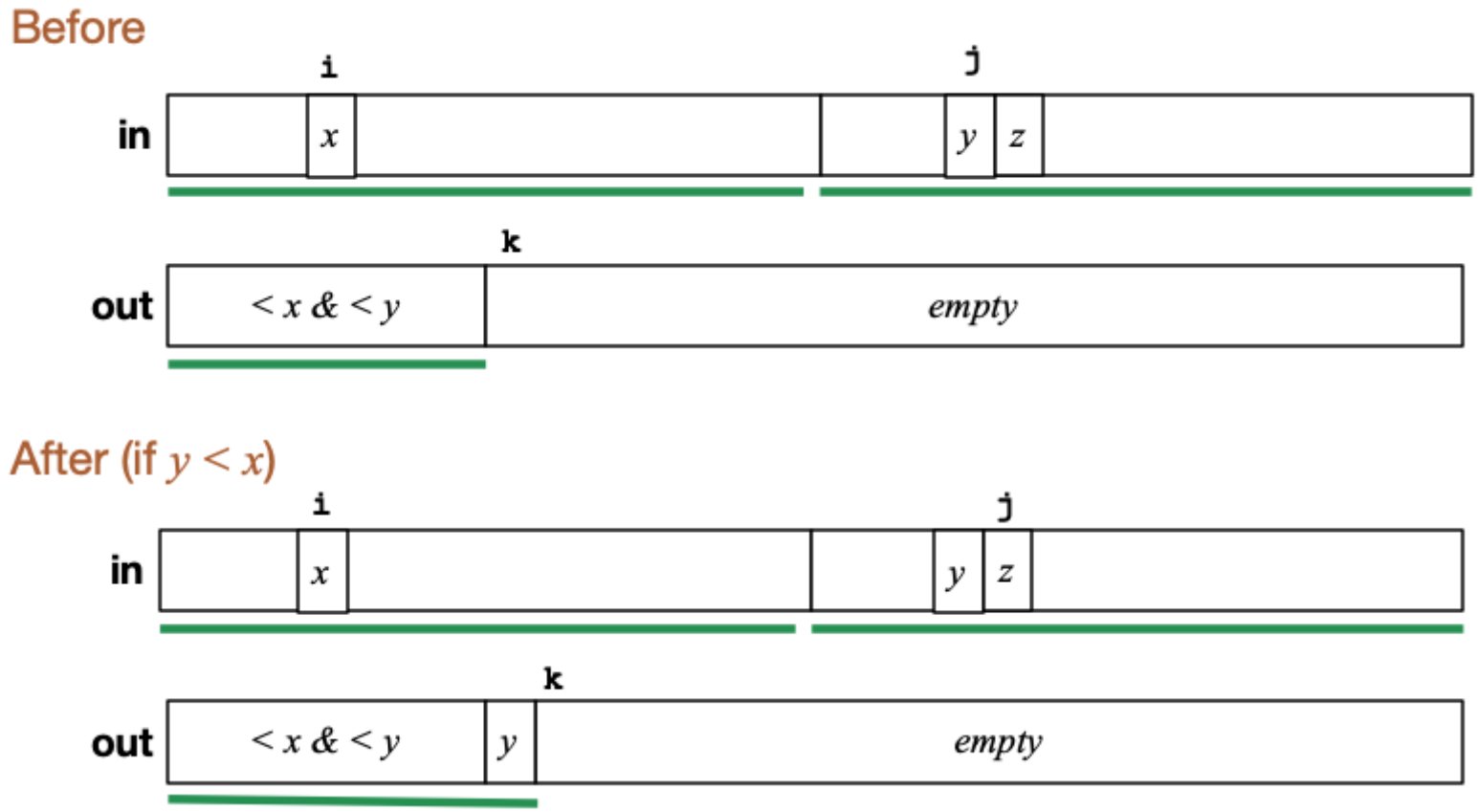
```
void mergesort(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2; // mid point
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
   merge(a, lo, mid, hi);
}
```

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

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

❖ ... Mergesort Implementation

One step in the merging process:



❖ ... Mergesort Implementation

Implementation of merge:

```
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) {
     if (less(a[i],a[j]))
        tmp[k++] = a[i++];
     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 ≤ *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

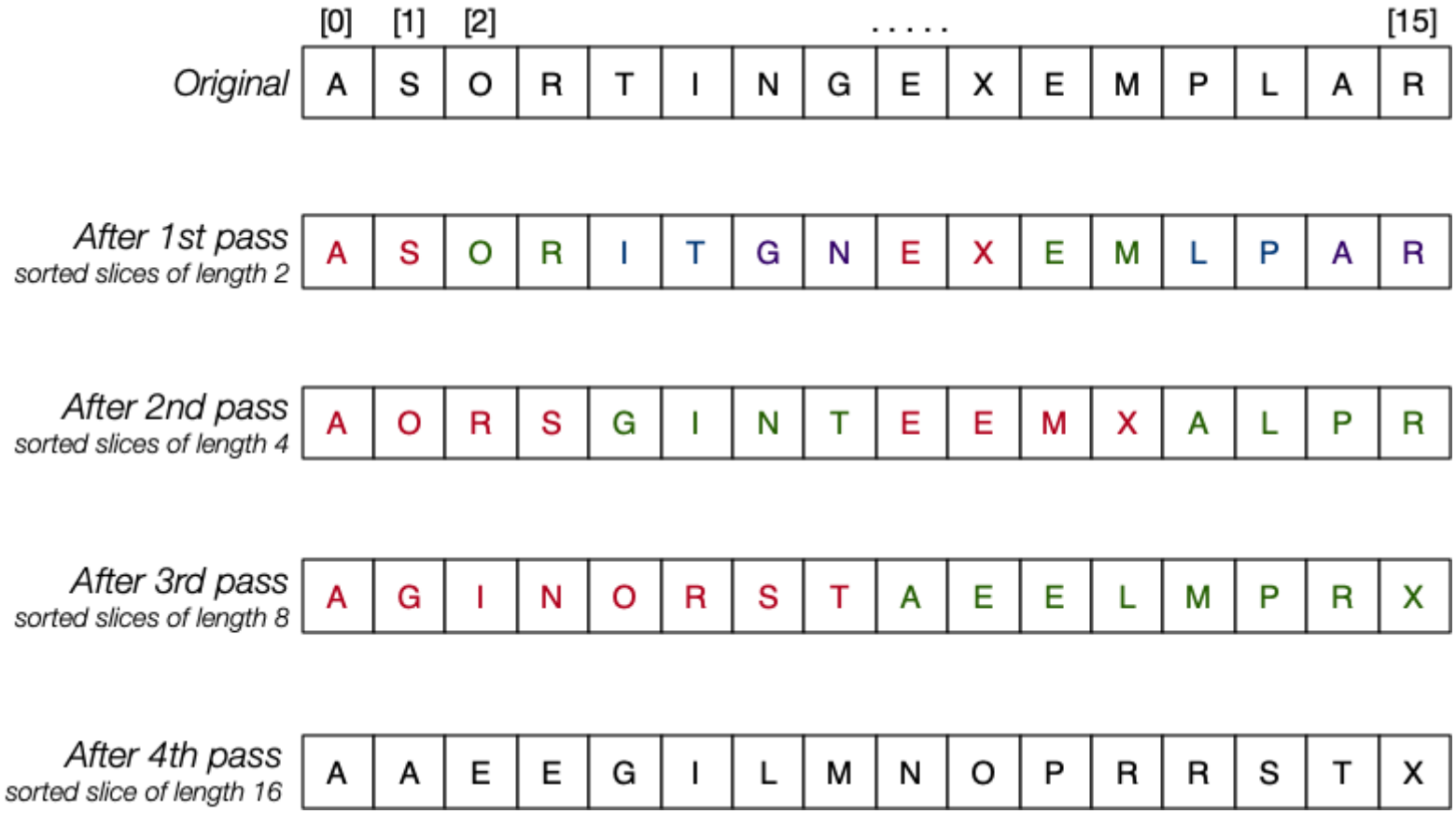
- partition boundaries can be computed iteratively

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



❖ ... 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);
      }
   }
}
```

❖ ... Bottom-up Mergesort

```
// 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[])
{
   int i; // index into a[]
   int j; // index into b[]
   int k; // index into c[]
   for (i = j = k = 0; k < aN+bN; k++) {
      if (i == aN)
         c[k] = b[j++];
      else if (j == bN)
         c[k] = a[i++];
      else if (less(a[i],b[j]))
         c[k] = a[i++];
      else
         c[k] = b[j++];
   }
}
```

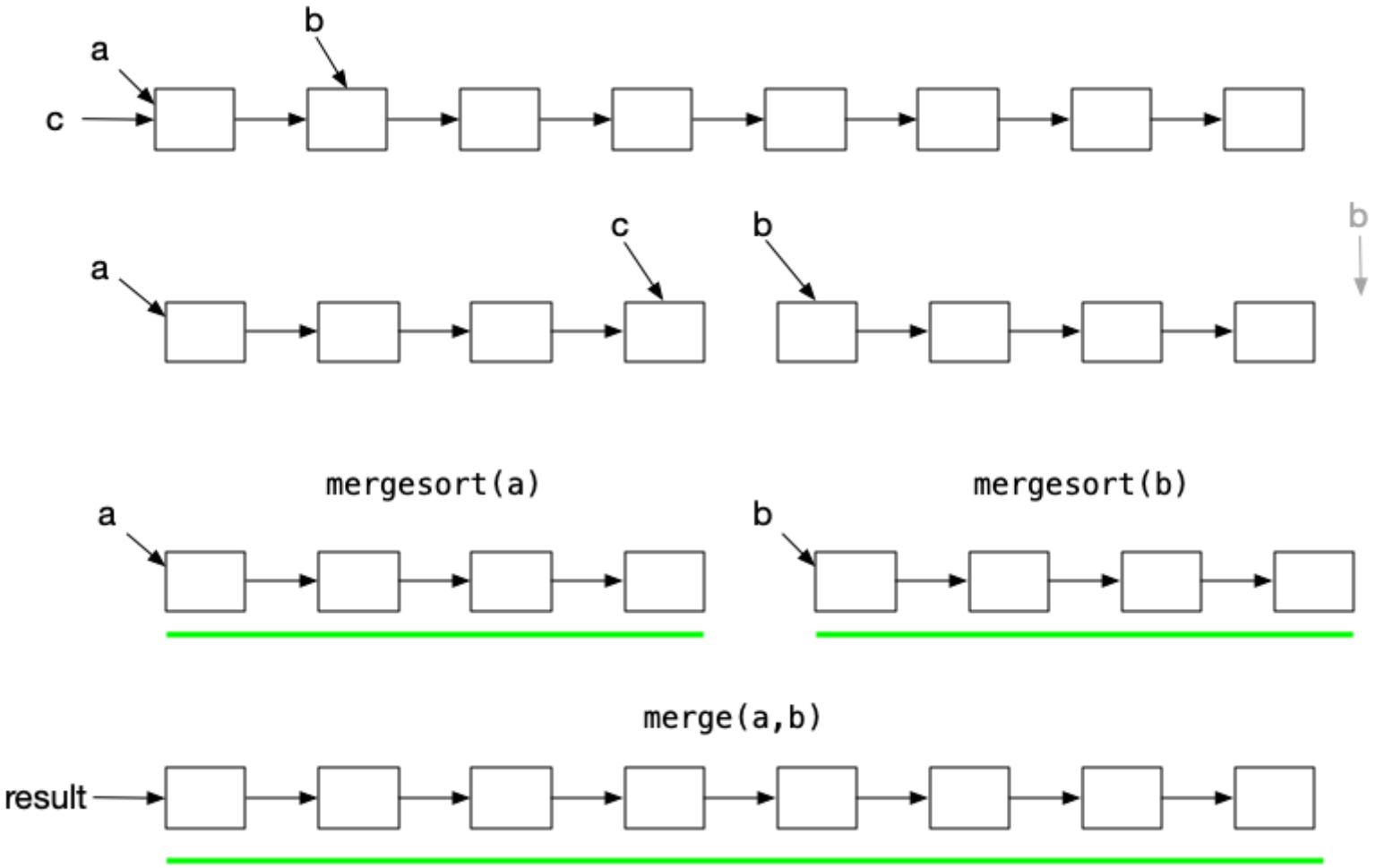
❖ Mergesort and Linked Lists

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
         { 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));
}
```

Produced: 21 Jul 2020