Sorting(2)

College of Computer Science, CQU

outline

- Shellsort
- Mergesort
- Quicksort

Shellsort (diminishing increment sort)

- Shellsort makes comparisons and swaps between non-adjacent elements
- Shellsort's strategy is to make the list "mostly sorted" so that a final Insertion Sort can finish the job.

Shellsort: Idea

- Shellsort breaks the array of elements into "virtual" sublists.
- Each sublist is sorted using an Insertion Sort.
- Another group of sublists is then chosen and sorted,
- and so on.

希尔排序----举例

初始关键字　49　38　65　97　76　13　27　49　55　4

49　13

38　27

65　49

97　55

76　4

一趟排序结果　13　27　49　55　4　49　38　65　97　76

13　55　38　76

27　4　65

49　49　97

二趟排序结果　13　4　49　38　27　49　55　65　97　76

三趟排序结果　4　13　27　38　49　49　55　65　76　97

Shellsort: Example

59 20 17 13 28 14 23 83 36 98 11 70 65 41 42 15

36 20 11 13 28 14 23 15 59 98 17 70 65 41 42 83

28 14 11 13 36 20 17 15 59 41 23 70 65 98 42 83

11 13 17 14 23 15 28 20 36 41 42 70 59 83 65 98

11 13 14 15 17 20 23 28 36 41 42 59 65 70 83 98

Shellsort: Implementation

```
// Modified version of Insertion Sort for varying increments
template <typename E, typename Comp>
void inssort2(E A[], int n, int incr) {
  for (int i=incr; i<n; i+=incr)
    for (int j=i; (j>=incr) &&
                  (Comp::prior(A[j], A[j-incr])); j-=incr)
      swap(A, j, j-incr);
}

template <typename E, typename Comp>
void shellsort(E A[], int n) { // Shellsort
  for (int i=n/2; i>2; i/=2)      // For each increment
    for (int j=0; j<i; j++)       // Sort each sublist
      inssort2<E,Comp>(&A[j], n-j, i);
  inssort2<E,Comp>(A, n, 1);
}
```

```
void ShellInsert ( E A[], int dk ) {
  for ( i=dk+1; i<=n; ++i )
    if ( A[i]< A[i-dk]) {
      A[0] = A[i];              // 暂存在A[0]
      for (j=i-dk;  j>0&&(A[0] <A[j]);
              j-=dk)
        A[j+dk] = A[j];  // 记录后移，查找插入位置
      A[j+dk] = A[0];                  // 插入
    } // if
} // ShellInsert
```

```
void ShellSort (E A[], int dlta[], int t)
{   // 增量为dlta[]的希尔排序
    for (k=0; k<t; ++t)
        ShellInsert(A, dlta[k]);
            //一趟增量为dlta[k]的插入排序
} // ShellSort
```

Shellsort: Complexity

- The average-case performance of Shellsort (for "divisions by three" increments) is $O(n^{1.5})$.
- Shellsort is substantially better than Insertion Sort, or any of exchange sorts $\Theta(n^2)$.

Mergesort

- Mergesort is one of the simplest sorting algorithms conceptually, and has good performance both in the asymptotic sense and in empirical running time.

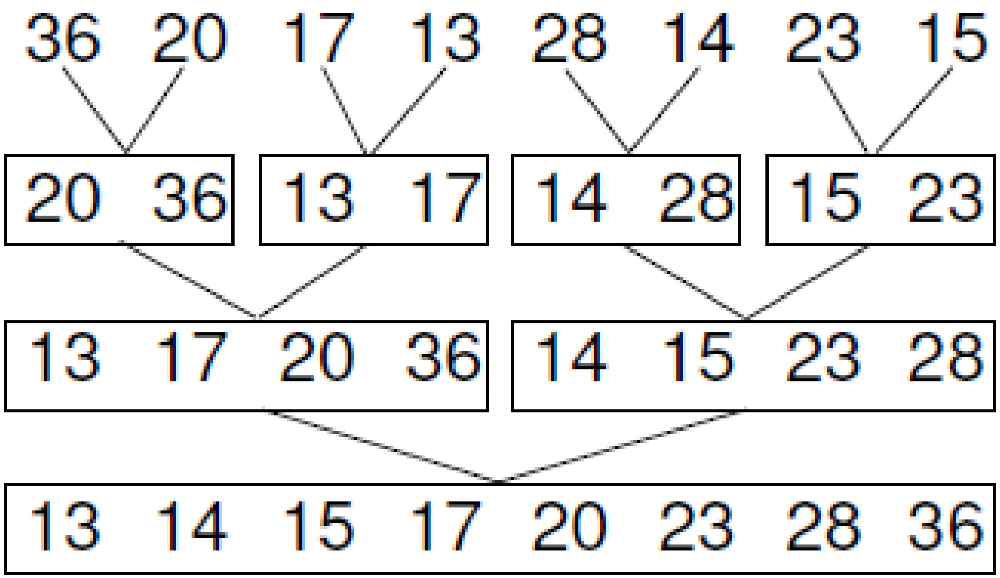
Mergesort: Idea

- **Idea: divide and conquer**
  - **Divide**: If S has at least two elements (nothing needs to be done if S has zero or one elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.
  - **Recur**: Recursive sort sequences S_1 and S_2.
  - **Conquer**: Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.

Mergesort: Pseudocode Sketch

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List L1 = half of the items from inlist;
  List L2 = other half of the items from inlist;
  return merge(mergesort(L1), mergesort(L2));
}
```

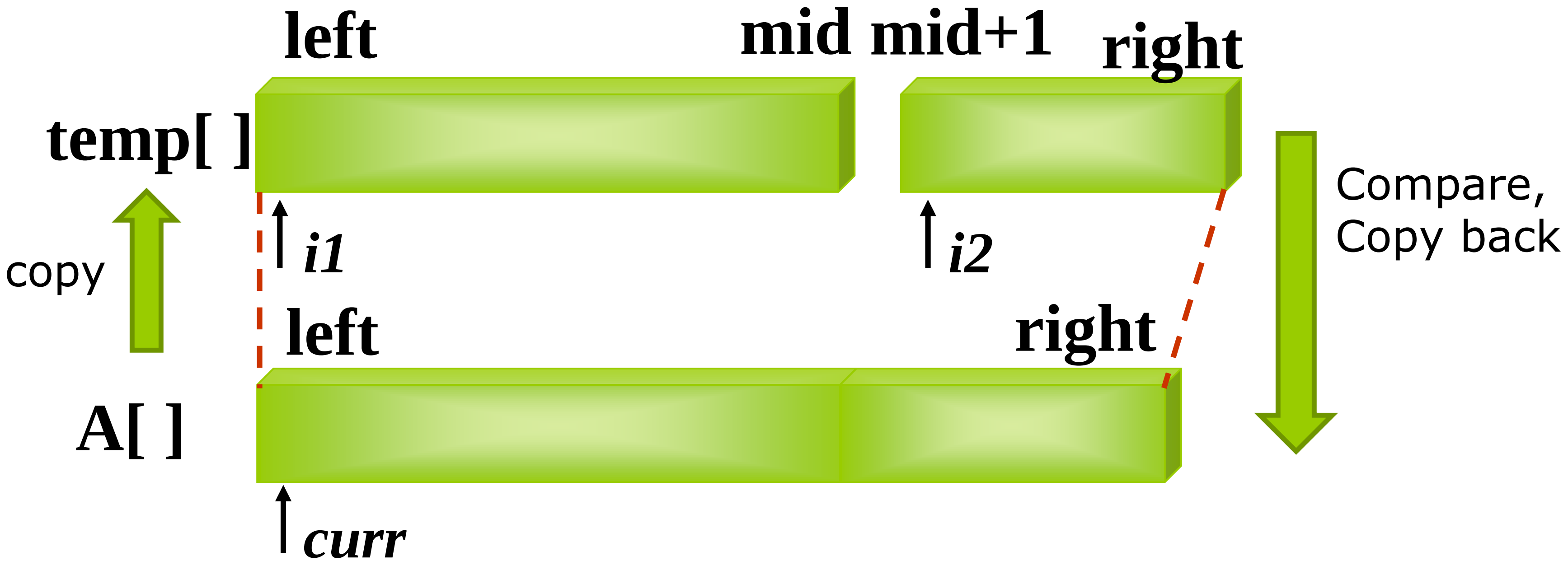
Mergesort: Example



Implementing Mergesort

- **how to represent the lists?**
- **linked list :**
  - Implementing merge for linked lists is straightforward, because we need only remove items from the front of the input lists and append items to the output list.
  - Breaking the input list into two equal halves presents some difficulty.
- **array:**
  - splitting input into two subarrays is easy if we know the array bounds.
  - Merging is also easy if we merge the subarrays into a second array.
  - a serious disadvantage: requires twice the amount of space as any of the sorting methods presented so far.

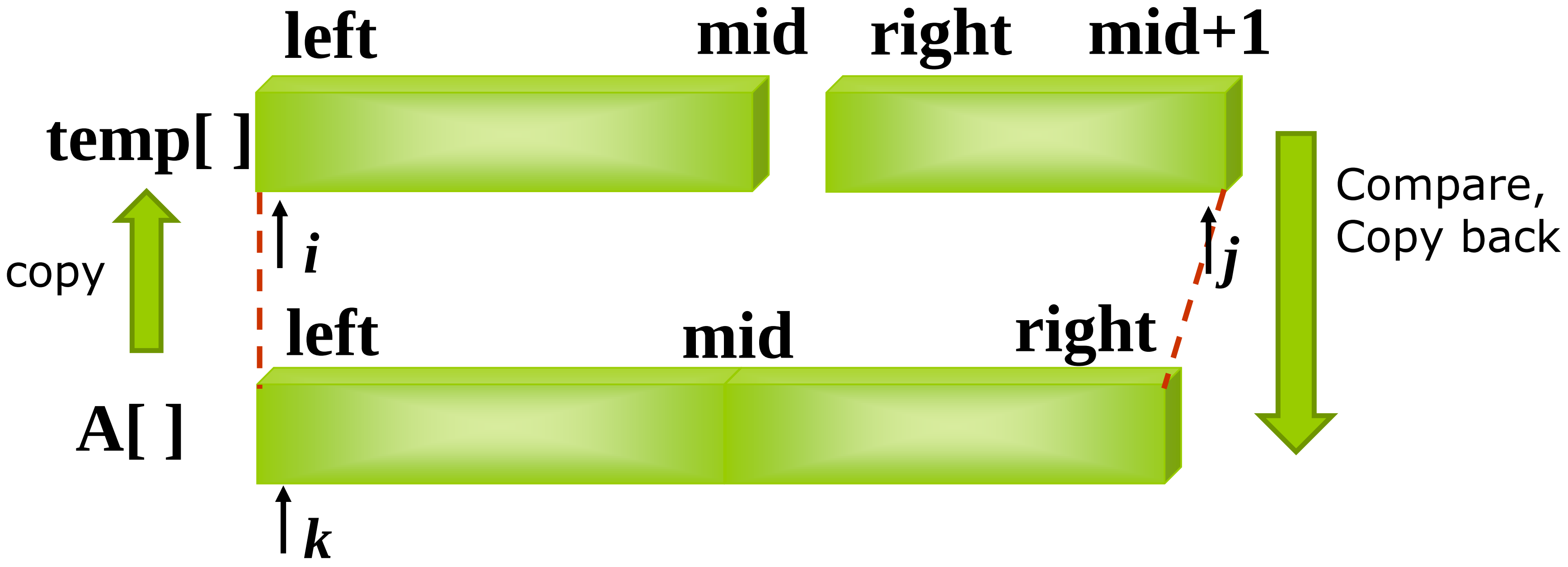
Merge (standard)



Mergesort: Standard Implementation

```
template <typename E, typename Comp>
void mergesort(E A[], E temp[], int left, int right) {
  if (left == right) return;         // List of one element
  int mid = (left+right)/2;
  mergesort<E,Comp>(A, temp, left, mid);
  mergesort<E,Comp>(A, temp, mid+1, right);
  for (int i=left; i<=right; i++)   // Copy subarray to temp
    temp[i] = A[i];
  // Do the merge operation back to A
  int i1 = left; int i2 = mid + 1;
  for (int curr=left; curr<=right; curr++) {
    if (i1 == mid+1)      // Left sublist exhausted
      A[curr] = temp[i2++];
    else if (i2 > right)  // Right sublist exhausted
      A[curr] = temp[i1++];
    else if (Comp::prior(temp[i1], temp[i2]))
      A[curr] = temp[i1++];
    else A[curr] = temp[i2++];
  }
}
```

Merge(optimized)



Mergesort: Optimized Implementation

```
template <typename E, typename Comp>
void mergesort(E A[], E temp[], int left, int right) {
  if ((right-left) <= THRESHOLD) { // Small list
    inssort<E,Comp>(&A[left], right-left+1);
    return;
  }
  int i, j, k, mid = (left+right)/2;
  mergesort<E,Comp>(A, temp, left, mid);
  mergesort<E,Comp>(A, temp, mid+1, right);
  // Do the merge operation.  First, copy 2 halves to temp.
  for (i=mid; i>=left; i--) temp[i] = A[i];
  for (j=1; j<=right-mid; j++) temp[right-j+1] = A[j+mid];
  // Merge sublists back to A
  for (i=left,j=right,k=left; k<=right; k++)
    if (Comp::prior(temp[i], temp[j])) A[k] = temp[i++];
    else A[k] = temp[j--];
}
```

Mergesort: Complexity

- **mergesort ‘s time complexity is** $\Theta(n \log n)$
- **it is the conceptually simplest algorithm we will see.**
- **Space Requirements for Mergesort**
  - two sorted linked lists without using any extra space.
  - However, to merge two sorted arrays (or portions of an array), we must use a third array to store the result of the merge.

Quicksort

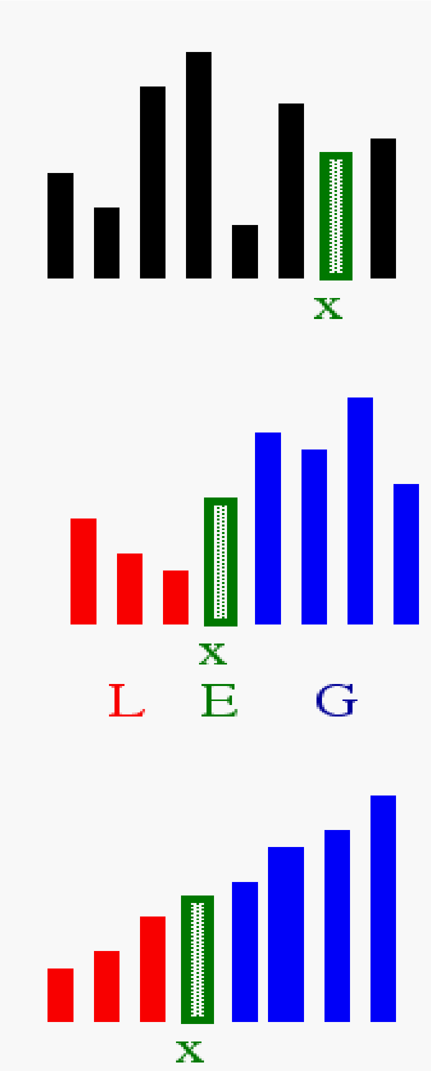
- Quicksort when properly implemented, is the fastest known general-purpose in-memory sorting algorithm in the average case.
- It does not require the extra array needed by Mergesort, so it is space efficient as well.
- Quicksort is widely used while hampered by exceedingly poor worst-case performance, thus making it inappropriate for certain applications.

Quicksort: Idea

1) Select: pick an pivot

2) Divide: rearrange elements so that x goes to its final position E

3) Recurse and Conquer: recursively sort



Quicksort Implementation

```
template <typename E, typename Comp>
void qsort(E A[], int i, int j) { // Quicksort
  if (j <= i) return; // Don't sort 0 or 1 element
  int pivotindex = findpivot(A, i, j);
  swap(A, pivotindex, j);    // Put pivot at end
  // k will be the first position in the right subarray
  int k = partition<E,Comp>(A, i-1, j, A[j]);
  swap(A, k, j);             // Put pivot in place
  qsort<E,Comp>(A, i, k-1);
  qsort<E,Comp>(A, k+1, j);
}
```

Findpivot()

```
template <typename E>
inline int findpivot(E A[], int i, int j)
   { return (i+j)/2; }
```

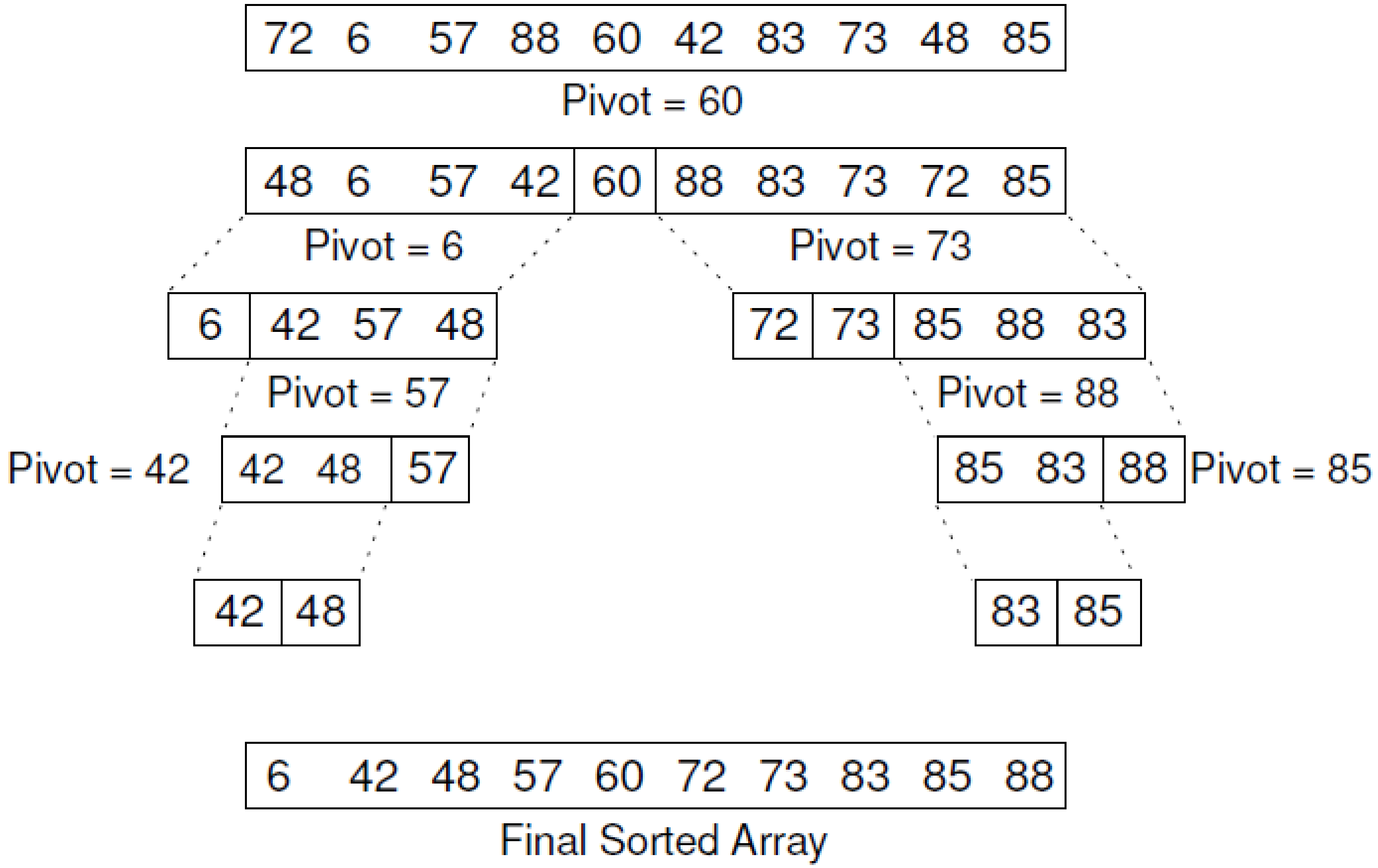
Partition()

```
template <typename E, typename Comp>
inline int partition(E A[], int l, int r, E& pivot) {
  do {              // Move the bounds inward until they meet
    while (Comp::prior(A[++l], pivot));  // Move l right and
    while ((l < r) && Comp::prior(pivot, A[--r])); // r left
    swap(A, l, r);              // Swap out-of-place values
  } while (l < r);              // Stop when they cross
  return l;      // Return first position in right partition
}
```

Partition: example

Initial		72	6	57	88	85	42	83	73	48	60
	l										r
Pass 1		72	6	57	88	85	42	83	73	48	60
		l								r	
Swap 1		48	6	57	88	85	42	83	73	72	60
		l								r	
Pass 2		48	6	57	88	85	42	83	73	72	60
					l		r				
Swap 2		48	6	57	42	85	88	83	73	72	60
					l		r				
Pass 3		48	6	57	42	85	88	83	73	72	60
						l,r					

Quicksort: Example



Quicksort: Complexity

- Worst case: when each pivot yields a bad partitioning of the array
  - Cost: $\Theta(n^2)$
- Best case: when findpivot always breaks the array into two equal halves
  - Cost: $\Theta(n \log n)$
- Average case: $\Theta(n \log n)$

References

- **Data Structures and Algorithm Analysis Edition 3.2 (C++ Version)**
  - P.239-248

-End-