Mergesort

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

Mergesort Overview

- Mergesort is a very popular sorting algorithm.
- The **worst-case** time complexity is $\Theta(N \log N)$.
  - Remember that quicksort has a worst-case complexity of $\Theta(N^2)$.
- Thus, mergesort is preferable if we want a theoretical guarantee that the time will never exceed $O(N \log N)$.
- Also, unlike quicksort, mergesort is **stable**.
- A sorting method is called **stable** if it does not change the order of items whose keys are equal.
- Disadvantage: mergesort typically requires extra $\Theta(N)$ space, to copy the data.
- Quicksort on the other hand sorts **in place:**
  - No extra memory, except for a constant amount for local variables.
  - This can be important, if you are sorting massive amounts of data.

Mergesort Implementation

- Top-level function:
- Allocates memory for a scratch array.
  - Note that the size of the scratch array is the same as the size of the input array A.
- Then, we just call a recursive helper function that does all the work.

```
void mergesort(Item * A, int length)
{
   Item * aux = malloc(sizeof(Item) * length);
   msort_help(A, length, aux);
   free(aux);
}

void msort_help(Item * A, int length, Item * aux)
{
   if (length <= 1) return;
   int M = length/2;
   Item * C = &(A[M]);
   int P = length-M;

   msort_help(A, M, aux);
   msort_help(C, P, aux);
    merge(A, A, M, C, P, aux);
}
```

Mergesort Implementation

- Recursive helper, basic approach:
- Split A to two halves.
- Left half:
  - Pointer to A.
  - Length M = length(A)/2.
- Right half:
  - Pointer C = &(A[M]).
  - Length P = length(A) - M.
- Sort each half.
  - Done via recursive call to itself.
- Merge the results (we will see how in a bit).

```
void mergesort(Item * A, int length)
{
    Item * aux = malloc(sizeof(Item) * length);
    msort_help(A, length, aux);
    free(aux);
}

void msort_help(Item * A, int length, Item * aux)
{
    if (length <= 1) return;
    int M = length/2;
    Item * C = &(A[M]);
    int P = length-M;

    msort_help(A, M, aux);
    msort_help(C, P, aux);
    merge(A, A, M, C, P, aux);
}
```

Merging Two Sorted Sets

- Summary:
  - Assumption: arrays B and C are already sorted.
  - Merges B and C, produces sorted array D:
- Result array pointer D is passed as input.
  - It must already have enough memory.

```
void merge(Item *D, Item *B, int M,
               Item *C, int P, Item * aux )
{
   int T = M+P;
   int i, j, k;

   for (i = 0; i < M; i++) aux[i] = B[i];
   for (j = 0; j < P; j++) aux[j+M] = C[j];

   for (i = 0, j = M, k = 0; k < T; k++)
   {
      if (i == M) { D[k] = aux[j++]; continue; }
      if (j == T) { D[k] = aux[i++]; continue; }
      if (less(aux[i], aux[j])) D[k] = aux[i++];
      else D[k] = aux[j++];
   }
}
```

Merging Two Sorted Sets

- Note that here is where we use the aux array.
- Why do we need it?

```
void merge(Item *D, Item *B, int M,
               Item *C, int P, Item * aux )
{
   int T = M+P;
   int i, j, k;

   for (i = 0; i < M; i++) aux[i] = B[i];
   for (j = 0; j < P; j++) aux[j+M] = C[j];

   for (i = 0, j = M, k = 0; k < T; k++)
   {
      if (i == M) { D[k] = aux[j++]; continue; }
      if (j == T) { D[k] = aux[i++]; continue; }
      if (less(aux[i], aux[j])) D[k] = aux[i++];
      else D[k] = aux[j++];
   }
}
```

Merging Two Sorted Sets

- Note that here is where we use the aux array.
- Why do we need it?
- Using aux allows for D to share memory with B and C.
- We first copy the contents of both B and C to aux.
- Then, as we write into D, possibly overwriting B and C, aux still has the data we need.

```
void merge(Item *D, Item *B, int M,
              Item *C, int P, Item * aux )
{
   int T = M+P;
   int i, j, k;

   for (i = 0; i < M; i++) aux[i] = B[i];
   for (j = 0; j < P; j++) aux[j+M] = C[j];

   for (i = 0, j = M, k = 0; k < T; k++)
   {
      if (i == M) { D[k] = aux[j++]; continue; }
      if (j == T) { D[k] = aux[i++]; continue; }
      if (less(aux[i], aux[j])) D[k] = aux[i++];
      else D[k] = aux[j++];
   }
}
```

Merge Execution

Array D:	position	0	1	2	3	4	5	6	7	8	9
	value										

Array aux:	position	0	1	2	3	4	5	6	7	8	9
	value										

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Initial state

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	3	4	5	6	7	8	9
	value										

Array aux:	position	0	1	2	3	4	5	6	7	8	9
	value	10	35	75	80	90					

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Step 1:

Copy B to aux.

Merge Execution

Array D:	position	k=0	1	2	3	4	5	6	7	8	9
	value										

Array aux:	position	i=0	1	2	3	4	j=5	6	7	8	9
	value	10	35	75	80	90	17	30	40	60	70

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Main loop initialization

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	k=1	2	3	4	5	6	7	8	9
	value	10									

Array aux:	position	0	i=1	2	3	4	j=5	6	7	8	9
	value	10	35	75	80	90	17	30	40	60	70

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	k=2	3	4	5	6	7	8	9
	value	10	17								

Array aux:	position	0	i=1	2	3	4	5	j=6	7	8	9
	value	10	35	75	80	90	17	30	40	60	70

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	k=3	4	5	6	7	8	9
	value	10	17	30							

Array aux:	position	0	i=1	2	3	4	5	6	j=7	8	9
	value	10	35	75	80	90	17	30	40	60	70

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	3	k=4	5	6	7	8	9
	value	10	17	30	35						

Array aux:	position	0	1	i=2	3	4	5	6	j=7	8	9
	value	10	35	75	80	90	17	30	40	60	70

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	3	4	k=5	6	7	8	9
	value	10	17	30	35	40					

Array aux:	position	0	1	i=2	3	4	5	6	7	j=8	9
	value	10	35	75	80	90	17	30	40	60	70

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	3	4	5	k=6	7	8	9
	value	10	17	30	35	40	60				

Array aux:	position	0	1	i=2	3	4	5	6	7	8	j=9
	value	10	35	75	80	90	17	30	40	60	70

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	3	4	5	6	k=7	8	9
	value	10	17	30	35	40	60	70			

Array aux:	position	0	1	i=2	3	4	5	6	7	8	9	j=10
	value	10	35	75	80	90	17	30	40	60	70	

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	3	4	5	6	7	k=8	9
	value	10	17	30	35	40	60	70	75		

Array aux:	position	0	1	2	i=3	4	5	6	7	8	9	j=10
	value	10	35	75	80	90	17	30	40	60	70	

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	3	4	5	6	7	8	k=9
	value	10	17	30	35	40	60	70	75	80	

Array aux:	position	0	1	2	3	i=4	5	6	7	8	9	j=10
	value	10	35	75	80	90	17	30	40	60	70	

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Merge Execution

Array D:	position	0	1	2	3	4	5	6	7	8	9
	value	10	17	30	35	40	60	70	75	80	90

k=10

Array aux:	position	0	1	2	3	4	i=5	6	7	8	9
	value	10	35	75	80	90	17	30	40	60	70

j=10

Array B:	position	0	1	2	3	4
	value	10	35	75	80	90

k=10, so we are done!

Array C:	Position	0	1	2	3	4
	Value	17	30	40	60	70

Mergesort Execution Trace

mergesort([35, 30, 17, 10, 60])

Mergesort Execution Trace

mergesort([35, 30, 17, 10, 60])

    mergesort([35, 30])

Mergesort Execution Trace

mergesort([35, 30, 17, 10, 60])

  mergesort([35, 30]) → [30,35]

    mergesort([35]) → [35]

    mergesort([30]) → [30]

    merge([35],[30]) → [30,35]

Mergesort Execution Trace

mergesort([35, 30, 17, 10, 60])

    mergesort([35, 30]) → [30,35]

        mergesort([35]) → [35]

        mergesort([30]) → [30]

        merge([35],[30]) → [30,35]

    mergesort([17, 10, 60])

        mergesort([17]) → [17]

        mergesort([10, 60])

Mergesort Execution Trace

mergesort([35, 30, 17, 10, 60])

    mergesort([35, 30]) →[30,35]

        mergesort([35]) → [35]

        mergesort([30]) → [30]

        merge([35],[30]) → [30,35]

    mergesort([17, 10, 60])

        mergesort([17]) → [17]

        mergesort([10, 60]) → [10, 60]

            mergesort([10]) → [10]

            mergesort([60]) → [60]

            merge([10],[60]) → [10,60]

Mergesort Execution Trace

mergesort([35, 30, 17, 10, 60])

  mergesort([35, 30]) → [30,35]

    mergesort([35]) → [35]

    mergesort([30]) → [30]

    merge([35],[30]) → [30,35]

  mergesort([17, 10, 60]) → [10,17,60]

    mergesort([17]) → [17]

    mergesort([10, 60]) → [10, 60]

      mergesort([10]) → [10]

      mergesort([60]) → [60]

      merge([10],[60]) → [10,60]

    merge([17], [10,60]) → [10,17,60]

Mergesort Execution Trace

mergesort([35, 30, 17, 10, 60]) → [10, 17, 30, 35, 60]

  mergesort([35, 30]) →[30,35]

    mergesort([35]) → [35]

    mergesort([30]) → [30]

    merge([35],[30]) → [30,35]

  mergesort([17, 10, 60]) → [10,17,60]

    mergesort([17]) → [17]

    mergesort([10, 60])

      mergesort([10]) → [10]

      mergesort([60]) → [60]

      merge([10],[60]) → [10,60]

    merge([17], [10,60]) → [10,17,60]

  merge([30, 35], [10,17,60]) → [10, 17, 30, 35, 60]

Mergesort Complexity

- Let T(N) be the **worst-case** running time complexity of mergesort for sorting N items.
- What is the recurrence for T(N)?

Mergesort Complexity

- Let T(N) be the **worst-case** running time complexity of mergesort for sorting N items.
- $T(N) = 2T(N/2) + N.$
- Let $N = 2^n$.
- $T(N) = 2T(2^{n-1}) + 2^n$

$$= 2^2T(2^{n-2}) + 2*2^n$$
$$= 2^3T(2^{n-3}) + 3*2^n$$
$$= 2^4T(2^{n-4}) + 4*2^n$$
$$\ldots$$
$$= 2^nT(2^{n-n}) + n*2^n$$
$$= N * \text{constant} + N * \lg N = \mathbf{\Theta(N \lg N).}$$

Mergesort Variations

- There are several variations of mergesort that the textbook provides, that may be useful in different cases:
- Mergesort with no need for linear extra space.
  - More complicated, somewhat slower.
- Bottom-up mergesort, without recursive calls.
- Mergesort using lists and not arrays.
- You should be aware of them, but it is not really worth going over them in class, they reuse the basic concepts that we have already talked about.