Divide and Conquer Sorting

CSE 326
Data Structures
"Divide and Conquer"

- Very important strategy applied to many computer science problems:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine solutions to get overall solution
“Divide and Conquer”

• Two divide and conquer sorting methods:

• **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves $\rightarrow$ known as **Mergesort**

• **Idea 2**: Partition array into small items and large items, then *recursively* sort the two smaller portions $\rightarrow$ known as **Quicksort**
Mergesort

- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together
• The merging requires an auxiliary array.

Auxiliary array

2 4 8 9 1 3 5 6

Auxiliary array
 Auxiliary Array

• The merging requires an auxiliary array.

2 4 8 9 1 3 5 6

Auxiliary array

1

Auxiliary array
Auxiliary Array

• The merging requires an auxiliary array.

Auxiliary array

| 2 | 4 | 8 | 9 | 1 | 3 | 5 | 6 |

Auxiliary array

| 1 | 2 |
Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

```
1 2 3   Auxiliary array
```
Auxiliary Array

- The merging requires an auxiliary array.

\[
\begin{array}{ccccccc}
2 & 4 & 8 & 9 & 1 & 3 & 5 & 6 \\
\end{array}
\]

Auxiliary array
Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

```
1 2 3 4 5
```

Auxiliary array
Auxiliary Array

- The merging requires an auxiliary array.

```
2 4 8 9 1 3 5 6
```

```
1 2 3 4 5 6
```
Auxiliary Array

- The merging requires an auxiliary array.

```
  2  4  8  9  1  3  5  6
```

```
  1  2  3  4  5  6  8
```

Auxiliary array
Auxiliary Array

- The merging requires an auxiliary array.

```plaintext
2 4 8 9 1 3 5 6
```

```plaintext
1 2 3 4 5 6 8 9
```
Mergesort Example

1. Divide
   - 8 2 9 4
   - 5 3 1 6

2. Divide
   - 8 2
   - 9 4
   - 5 3
   - 1 6

3. 1 element
   - 8 2
   - 9 4
   - 5 3
   - 1 6

4. Merge
   - 2 8
   - 4 9

5. Merge
   - 2 4 8 9
   - 1 3 5 6
Typical Merging

**Normal**

**Left completed first**

Diagram showing the typical merging process with indices i and j, and target and copy regions.
Left Side Completes First
Right Side Completes First
Recursive Mergesort

MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort[A,T,1,n];
}

Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A,T,left,mid);
    Mergesort(A,T,mid+1,right);
    Merge(A,T,left,right);
}
Merge(A[], T[] : integer array, left, right : integer) : {
    mid, i, j, k, l, target : integer;
    mid := (right + left)/2;
    i := left; j := mid + 1; target := left;
    while i < mid and j < right do
        if A[i] < A[j]
            then T[target] := A[i]; i := i + 1;
        else T[target] := A[j]; j := j + 1;
        target := target + 1;
    if i > mid then //left completed//
        for k := left to target-1 do A[k] := T[k];
    if j > right then //right completed//
        k := mid; l := right;
        while k > i do A[l] := A[k]; k := k-1; l := l-1;
        for k := left to target-1 do A[k] := T[k];
}
Merging

Merge(A[], T[], left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;
  i := left; j := mid + 1; target := left;
  while i <= mid and j <= right do
    if A[i] <= A[j]
      then T[target] := A[i]; i := i + 1;
      else T[target] := A[j]; j := j + 1;
      target := target + 1;
  if i > mid then //left completed//
    for k := left to target-1 do A[k] := T[k];
  if j > right then //right completed//
    k := mid; l := right;
    while k > i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
}
Merging

Merge(A[], T[] : integer array, left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;
  i := left; j := mid + 1; target := left;
  while i < mid and j < right do
    if A[i] < A[j]
      then T[target] := A[i]; i := i + 1;
    else T[target] := A[j]; j := j + 1;
    target := target + 1;
  if i > mid then //left completed//
    for k := left to target-1 do A[k] := T[k];
  if j > right then //right completed//
    k := mid; l := right;
    while k > i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
}

Merging

Merge(A[], T[] : integer array, left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;
  i := left; j := mid + 1; target := left;
  while i < mid and j < right do
    if A[i] < A[j] then
      T[target] := A[i]; i := i + 1;
    else T[target] := A[j]; j := j + 1;
    target := target + 1;
  if i > mid then //left completed//
    for k := left to target-1 do A[k] := T[k];
  if j > right then //right completed//
    k := mid; l := right;
    while k > i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
}
Merging

Merge(A[], T[] : integer array, left, right : integer) : {
    mid, i, j, k, l, target : integer;
    mid := (right + left)/2;
    i := left; j := mid + 1; target := left;
    while i \leq mid and j \leq right do
        if A[i] \leq A[j]
            then T[target] := A[i]; i := i + 1;
        else T[target] := A[j]; j := j + 1;
        target := target + 1;
    if i > mid then //left completed//
        for k := left to target-1 do A[k] := T[k];
    if j > right then //right completed//
        k := mid; l := right;
        while k > i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
}
Merging

Merge(A[], T[] : integer array, left, right : integer) : {
    mid, i, j, k, l, target : integer;
    mid := (right + left)/2;
    i := left; j := mid + 1; target := left;
    while i ≤ mid and j ≤ right do
        if A[i] ≤ A[j]
            then T[target] := A[i] ; i:= i + 1;
               else T[target] := A[j]; j := j + 1;
               target := target + 1;
        if i > mid then //left completed//
            for k := left to target-1 do A[k] := T[k];
        if j > right then //right completed//
            k := mid ; l := right;
            while k > i do A[l] := A[k]; k := k-1; l := l-1;
            for k := left to target-1 do A[k] := T[k];
    }
Mergesort Analysis

• Let $T(N)$ be the running time for an array of $N$ elements

• Mergesort divides array in half and calls itself on the two halves. After these recursive calls complete, it merges both halves using a temporary array

• Each recursive call takes $T(N/2)$ and merging takes $O(N)$
Mergesort Recurrence Relation

- The recurrence relation for $T(N)$ is:
  - $T(1) \leq c$
    - base case: 1 element array → constant time
  - $T(N) \leq 2T(N/2) + dN$
    - Sorting $n$ elements takes
      - the time to sort the left half
      - plus the time to sort the right half
      - plus an $O(N)$ time to merge the two halves

- $T(N) = O(N \log N)$
Ideas for Obvious Improvement?

- Half our copies are wasted
- Cleaning up the temporary storage:
Iterative Mergesort

Merge by 1
Merge by 2
Merge by 4
Merge by 8
Merge by 16
↓ Copy if Needed
Iterative pseudocode

• Sort(array A of length N)
  › Let \( m = 2 \), let B be temp array of length N
  › While \( m < N \)
    • For \( i = 1 \ldots N \) in increments of \( m \)
      – merge \( A[i \ldots i+m/2] \) and \( A[i+m/2 \ldots i+m] \) into \( B[i \ldots i+m] \)
    • Swap role of A and B
    • \( m = m \times 2 \)
  › If needed, copy B back to A
Properties of Mergesort

• Not in-place
  › Requires an auxiliary array

• Iterative Mergesort reduces copying
Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does

  › Partition array into left and right sub-arrays
    • the elements in left sub-array are all less than pivot
    • elements in right sub-array are all greater than pivot

  › Recursively sort left and right sub-arrays

  › Concatenate left and right sub-arrays in O(1) time
The steps of QuickSort

1. Select pivot value
2. Partition S
3. QuickSort(S₁) and QuickSort(S₂)
4. Presto! S is sorted

[Weiss]
“Four easy steps”

• To sort an array \( S \)
  › If the number of elements in \( S \) is 0 or 1, then return. The array is sorted.
  › Pick an element \( v \) in \( S \).
    This is the \textit{pivot} value.
  › Partition \( S\setminus\{v\} \) into two disjoint subsets, \( S_1 = \{ \text{all values } \leq v \} \), and \( S_2 = \{ \text{all values } \geq v \} \).
  › Return QuickSort\((S_1)\), \( v \), QuickSort\((S_2)\)
Quicksort Example

Divide

Divide

Divide

1 element

Conquer

Conquer

Conquer

Conquer
Details, details

- Picking the pivot
  - want a value that causes $|S_1|$ and $|S_2|$ to be non-zero and close to equal in size

- Implementing the actual partitioning

- Dealing with cases where elements are equal to the pivot
Potential Pivot Rules

• Chose A[left]
  › Fast, but too biased, enables worst-case

• Chose A[random], left ≤ random ≤ right
  › Completely unbiased
  › Will cause relatively even split, but slow

• Median of three, A[left], A[right], A[(left+right)/2]
  › A common approach, tends to be unbiased, and does a little sorting on the side.
Quicksort Partitioning

- Need to partition the array into left and right
  - the elements in left sub-array are \( \leq \) pivot
  - elements in right sub-array are \( \geq \) pivot

- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions
Partitioning Done In-Place

- One implementation (there are others)
  - median3 finds pivot and sorts left, center, right
  - Swap pivot with next to last element
  - Set pointers i and j to start and end of array
  - Repeat until i and j cross
    - Increment i until you hit element A[i] > pivot
    - Decrement j until you hit element A[j] < pivot
    - Swap A[i] and A[j]
  - Swap pivot (= A[N-2]) with A[i]
Choose the pivot as the median of three.

Place the pivot and the largest at the right and the smallest at the left.
Example

Move i to the right to be larger than pivot.

Move j to the left to be smaller than pivot.

Swap
Example

\[
\begin{array}{ccccccc}
0 & 1 & 4 & 2 & 7 & 3 & 5 & 9 & 6 & 8 \\
\end{array}
\]

Move i

\[
\begin{array}{ccccccc}
0 & 1 & 4 & 2 & 7 & 3 & 5 & 9 & 6 & 8 \\
\end{array}
\]

Move j

\[
\begin{array}{ccccccc}
0 & 1 & 4 & 2 & 7 & 3 & 5 & 9 & 6 & 8 \\
\end{array}
\]

Swap

\[
\begin{array}{ccccccc}
0 & 1 & 4 & 2 & 5 & 3 & 7 & 9 & 6 & 8 \\
\end{array}
\]

Move i

\[
\begin{array}{ccccccc}
0 & 1 & 4 & 2 & 5 & 3 & 7 & 9 & 6 & 8 \\
\end{array}
\]

Move j

\[
\begin{array}{ccccccc}
0 & 1 & 4 & 2 & 5 & 3 & 7 & 9 & 6 & 8 \\
\end{array}
\]

Swap Pivot

\[
\begin{array}{ccccccc}
0 & 1 & 4 & 2 & 5 & 3 & 6 & 7 & 9 & 8 \\
\end{array}
\]

\(S_1 < \text{pivot}\) \hspace{1cm} \text{pivot} \hspace{1cm} \(S_2 > \text{pivot}\)
Quicksort Best Case Performance

• Algorithm always chooses best pivot and splits sub-arrays in half
  
  › T(0) = T(1) = O(1)
    • constant time if 0 or 1 element
  
  › For N > 1,
    2 recursive calls plus linear partitioning
  
  › T(N) = 2T(N/2) + O(N)
    • Same recurrence relation as Mergesort
  
  › T(N) = $O(N \log N)$
QuickSort Worst Case Performance

- Algorithm always chooses the worst pivot, one sub-array is empty at each recursion
  - $T(N) \leq a$ for $N \leq C$
  - $T(N) \leq T(N-1) + bN$
  - $\leq T(N-2) + b(N-1) + bN$
  - $\leq T(C) + b(C+1)+ \ldots + bN$
  - $\leq a + b(C + C+1 + C+2 + \ldots + N)$
  - $T(N) = O(N^2)$

- Fortunately, *average case* is $O(N \log N)$
  (see text for proof)
Properties of Quicksort

• No iterative version (without using an explicit stack).
• “In-place”, but uses auxiliary storage because of recursive calls.
• $O(n \log n)$ average case performance, but $O(n^2)$ worst case performance.
Opportunity for Improvement

- Applies for only small N
- Overhead of recursion starts to dominate
- Apply insertion sort for small N
Recursive Quicksort with Cutoff

Quicksort(A[]): integer array, left, right : integer): {
    pivotindex : integer;
    if left + CUTOFF ≤ right then
        pivot := median3(A, left, right);
        pivotindex := Partition(A, left, right-1, pivot);
        Quicksort(A, left, pivotindex - 1);
        Quicksort(A, pivotindex + 1, right);
    else
        Insertionsort(A, left, right);
}

CUTOFF = 10 is reasonable.
So Which Is Best?

• It’s a trick question, a naïve question
• Myth: “Quicksort is the best in-memory sorting algorithm.”
• Mergesort and Quicksort make different tradeoffs regarding the cost of comparison and the cost of a swap
• Mergesort is also the basis for external sorting algorithms (large N sorting)