Divide and Conquer Algorithms
Drawing a Ruler

Problem: Draw marks at regular intervals on a line

Write recursive

\textit{drawRuler} (xBegin, xEnd, height)

assuming

\textit{drawMark} (xCoord, height) is written

and a height of 0 indicates no further divisions
Recursive Ruler

• `drawRuler (begin, end, h)`
  • Base case?
  • ...

• A *Divide and Conquer (and Combine)* algorithm
  • Split problem into subproblems
  • Solve subproblems
  • Combine solutions to subproblems to solve whole problem
Iterative Ruler

void
drawRuler (begin, end, incr)
{
    // Start edge
drawMark (begin, END_HEIGHT);
begin += incr;
while (begin != end)
{
    drawMark (begin, ?);
    begin += incr;
}
    // End edge
drawMark (end, END_HEIGHT);
}
Two Sorting Algorithms

• Divide and Conquer Sorting Algorithms
  • Merge Sort
    • Split in half (divide)
    • Sort halves recursively (conquer)
    • Merge sorted halves (combine)
    • \(Worst = Average = O(N \lg N)\)

• Quick Sort
  • Choose pivot value from array
  • Place pivot in final position (divide)
    • Elements to left \(\leq\) pivot
    • Elements to right \(\geq\) pivot
  • Sort 2 sublists recursively (conquer)
  • Average = \(O(N \lg N)\)
  • \(Worst = O(N^2)\)
void mergeSort (int A[], size_t first, size_t last)
{
    if (last - first > 1) {
        size_t mid = first + (last - first) / 2;
        mergeSort (A, first, mid);
        mergeSort (A, mid, last);
        inplace_merge (A + first, A + mid, A + last);
    }
}
Merge Algorithm

• Merge sort requires *merge* algorithm
  • Complexity of merge?

• Out-of-place merge: merge (v, first, mid, last)
  • v[first, mid) sorted
  • v[mid, last) sorted

• Result: v[first, last) sorted
Merge Algorithm (Cont’d)
Merge Algorithm (Cont’d)
Merge Algorithm (Cont’d)

sublist A

sublist B

indexA

indexB

tempVector

last

first

last
Partitioning and Merging of Sublists in Merge Sort

Postorder algorithm
Call Tree for Merge Sort

Call msort()
- recursive call_1 to msort()
- recursive call_2 to msort()
- call_1 merge()

msort() : n/2

msort() : n/4

msort() : n/8

Level_0:

Level_1:

msort() : n/2

msort() : n/4

msort() : n/8

msort() : n/4

Level_2:

msort() : n/8

msort() : n/8

msort() : n/8

Level_3:

msort() : n/8

msort() : n/8

msort() : n/8

Level_i:

...
Quicksort Code

void quickSort (int A[], size_t first, size_t last)
{
    if (last - first > 1) {
        auto i = partition (A, first, last);
        // Pivot is placed in A[i]
        quickSort (A, first, i);
        quickSort (A, i + 1, last);
    }
}

**Partition Routine**

// will go into details later...

```c
size_t partition (int A[], size_t first, size_t last) {
    median3 (A, first, last - 1);
    int pivot = A[last - 2];
    size_t up = first, down = last - 2;
    for (; ; ) {
        while (A[++up] < pivot) {}  
        while (A[--down] > pivot) {}  
        if (up >= down) break;
        swap (A[up], A[down]);
    }
    swap (A[last - 2], A[up]);
    return up;
}
```
Quicksort

• *Fastest* known sorting algorithm in practice
  • Part of std::sort

  • May be used in cstdlib qsort
    • qsort (void* base, size_t num, size_t size,
      int (*comp)(const void*, const void*))

• Average case: O(N log N)

• *Worst case*: O(N^2)
Quick Sort

• Divide step
  • Pick any element (pivot) \( v \) in \( S \)
  • Partition \( S - \{ v \} \) into two disjoint groups
    \[ S_1 = \{ x \in S - \{ v \} \mid x \leq v \} \]
    \[ S_2 = \{ x \in S - \{ v \} \mid x \geq v \} \]

• Conquer step: recursively sort \( S_1 \) and \( S_2 \)

• Combine step: list the sorted \( S_1 \), followed by \( v \), followed by sorted \( S_2 \)
Quick Sort...
Quick Sort...
Partitioning

- Partitioning
  - Key step of quicksort algorithm
  - Many ways to implement

- How to pick pivot will be discussed later
Partitioning Strategy

• Want to partition an array A[left .. right]

• Swap pivot and A[right]

• Let $i = \text{left}; \ j = \text{right} - 1$
Partitioning Strategy

• Want to have
  • $A[k] \leq \text{pivot}, \text{ for } k < i$
  • $A[k] \geq \text{pivot}, \text{ for } k > j$

• While $i < j$
  • Move $i$ right, skipping over elements smaller than pivot
  • Move $j$ left, skipping over elements greater than pivot
  • When both $i$ and $j$ have stopped
    • $A[i] \geq \text{pivot}$
    • $A[j] \leq \text{pivot}$
Partitioning Strategy

• When i and j have stopped and i is to the left of j
  • Swap A[i] and A[j]

  • After swapping
    • A[i] <= pivot
    • A[j] >= pivot

  • Repeat the process until i and j cross

\[
\begin{array}{cccccc}
5 & 6 & 4 & 19 & 3 & 12 \\
\uparrow & & & & & \text{swap} \\
5 & 3 & 4 & 19 & 6 & 12 \\
\end{array}
\]
Partitioning Strategy

• When $i$ and $j$ have crossed
  • Swap $A[i]$ and pivot

• Result
  • $A[p] \leq pivot$, for $p < i$
  • $A[p] \geq pivot$, for $p > i$
Small arrays

• Cutoff value for small arrays

• Depends on
  • Time to make a recursive call
  • Architecture
  • Compiler
  • Other factors
Picking the Pivot

• Use the first element as pivot
  • OK if the input is random
  • What happens if the input is already sorted?
    • Can result in $O(N^2)$ behavior

• Choose the pivot randomly
  • Generally safe
  • Random number generation can be expensive
Picking the Pivot

• Use the median of the array

  • Partitioning always cuts the array into roughly half

• An *optimal* quicksort (O(N log N))

• How do you find the median?
Pivot: Median of Three

- Compare just three elements: the leftmost, rightmost and center
- Swap these elements if necessary so that
  - \( A[\text{left}] = \text{Smallest} \)
  - \( A[\text{right}] = \text{Largest} \)
  - \( A[\text{center}] = \text{Median of three} \)
- Pick \( A[\text{center}] \) as the pivot
- Swap \( A[\text{center}] \) and \( A[\text{right} - 1] \) so that pivot is at second-to-last position

```c
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

// Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```
Pivot: median of three


Swap A[center] and A[right]

Choose A[center] as pivot

Swap pivot and A[right – 1]

Only need to partition A[left + 1, ..., right – 2]. Why?
Main Quicksort Routine

```c
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

    // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }
    swap( a[ i ], a[ right - 1 ] ); // Restore pivot
    quicksort( a, left, i - 1 ); // Sort small elements
    quicksort( a, i + 1, right ); // Sort large elements
}
else // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```
Partitioning Part

• Works only if pivot is picked as *median-of-three*

  • \(A[\text{left}] \leq \text{pivot} \) and \(A[\text{right}] \geq \text{pivot}\)

  • Thus, only need to partition \(A[\text{left} + 1, \ldots, \text{right} - 2]\)

• \(j\) will not run past the end
  • because \(A[\text{left}] \leq \text{pivot}\)

• \(i\) will not run past the end
  • because \(A[\text{right}-1] = \text{pivot}\)

```c
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```
Quicksort vs. Mergesort

• quicksort and mergesort are $O(N \log N)$ in the average case

• Why is quicksort *faster* than mergesort?
  • Inner loop

• No extra juggling as in mergesort

```c
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) {}
    while( pivot < a[ --j ] ) {}
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```
Analysis

• Assumptions
  • A random pivot (no median-of-three partitioning)
  • No cutoff for small arrays

• Running time
  • pivot selection: constant time $O(1)$
  • partitioning: linear time $O(N)$
  • running time of two recursive calls

• $T(N) = T(i) + T(N - i - 1) + cN$ where $c$ is a constant
  • $i$: number of elements in $S1$
Worst-Case Analysis

• What will be the worst case?
  • Pivot is smallest element, all the time
  • Partition is always unbalanced

\[
T(N) = T(N-1) + cN \\
T(N-1) = T(N-2) + c(N-1) \\
T(N-2) = T(N-3) + c(N-2) \\
\vdots \\
T(2) = T(1) + c(2) \\
T(N) = T(1) + c \sum_{i=2}^{N} i = O(N^2)
\]
Best-case Analysis

• What will be the best case?
  • Partition is perfectly balanced
  • Pivot is always in the middle (median of the array)

\[
\begin{align*}
T(N) &= 2T(N/2) + cN \\
T(N) &= T(N/2) + c \\
T(N/2) &= T(N/4) + c \\
T(N/4) &= T(N/8) + c \\
&\vdots \\
T(2) &= T(1) + c \\
T(N) &= T(1) + c \log N \\
T(N) &= cN \log N + N = O(N \log N)
\end{align*}
\]
Efficiency of Quick Sort

• Best and worst case recurrences
  • Another way to solve them

• Best
  • \( T(N) = 2 \times T(N/2) + N; \ T(1) = 1 \)
  • \( T(N) = 2 \times (2 \times T(N/4) + N/2) + N \)
    = \( 2 \times (2 \times (2 \times T(N/8) + N/4) + N/2) + N \)
    = \( 2^k \times T(N/2^k) + k \times N \)
  • \( k = \log(N) \rightarrow T(N) = N \times T(1) + \log(N) \times N \)
    = \( O(N \log(N)) \)

• Worst
  • \( T(N) = T(N-1) + N; \ T(1) = 1 \)
  • \( T(N) = ? \)
Average-Case Analysis

• Difficult

• Average running time is $O(N \lg N)$
Finding $k^{th}$ Smallest Element

• Compute $k^{th}$ order statistic
  • $k = 1$ is min
  • $k = N$ is max

• Partition based on some pivot value
  • $i = \text{partition} \ (A, \ \text{left, right})$
  • if $i$ matches $k$ return $A[i]$  // Assuming 1-based
  • Recurse on left if $k < i$
  • Recurse on right if $k > i$
  • Find $(k - i)^{th}$ smallest

| values $\leq k^{th}$Smallest | $k^{th}$Smallest | values $\geq k^{th}$Smallest |