08 A: Sorting III

CS1102S: Data Structures and Algorithms

Martin Henz

March 10, 2010

Generated on Wednesday 10th March, 2010, 09:36

1. Recap: Sorting Algorithms
2. Analysis of Mergesort
3. Quicksort
4. A General Lower Bound for Sorting

1 Recap: Sorting Algorithms
- Sorting
- Insertion Sort
- Shell Sort
- Heapsort
- Mergesort

2 Analysis of Mergesort

3 Quicksort

4 A General Lower Bound for Sorting

Sorting

Input

Unsorted array of elements

Behavior

Rearrange elements of array such that the smallest appears first, followed by the second smallest etc, finally followed by the largest element

Comparison-based Sorting

The only requirement

A comparison function for elements

The only operation

Comparisons are the only operations allowed on elements (compare with Bucket Sort at the end of the lecture)

Insertion Sort: Idea

Passes

Algorithm proceeds in $N - 1$ *passes*

Invariant

After pass i, the elements in positions 0 to i are sorted.

Consequence of Invariant

After $N - 1$ passes, the whole array is sorted.

Shell Sort: Idea

Main idea

Proceed in passes $h_1, h_2, \ldots, h_t$, making sure that after each pass, $a[i] \leq a[i + h_k]$.

Invariant

After pass h_k, elements are still h_{k+1} sorted

Analysis

Shell's Increments

The worst-case running time of Shellsort, using Shell's increments $1, 2, 4, \ldots,$, is $\Theta(N^2)$.

Hibbards's Increments

The worst-case running time of Shellsort, using Hibbard's increments $1, 3, 7, \ldots, 2^k - 1$, is $\Theta(N^{3/2})$.

Heapsort: Idea

Use heap to sort

- Build heap from unsorted array (using percolateDown)

Heapsort: Idea

Use heap to sort

- Build heap from unsorted array (using percolateDown)
- Repeatedly take maximal element (using deleteMax)

Heapsort: Idea

Use heap to sort

- Build heap from unsorted array (using percolateDown)
- Repeatedly take maximal element (using deleteMax)

Reuse memory

Use free memory at the end of the heap in order to store the maximal element, leading to sorted array.

Mergesort: Idea

- Split unsorted arrays into two halves

Mergesort: Idea

- Split unsorted arrays into two halves
- Sort the two halves

Mergesort: Idea

- Split unsorted arrays into two halves
- Sort the two halves
- Merge the two sorted halves

Merging Two Sorted Arrays

- Use two pointers, one for each sorted array

Merging Two Sorted Arrays

- Use two pointers, one for each sorted array
- Compare values at pointer positions

Merging Two Sorted Arrays

- Use two pointers, one for each sorted array
- Compare values at pointer positions
    - Copy the smaller values into sorted array

Merging Two Sorted Arrays

- Use two pointers, one for each sorted array
- Compare values at pointer positions
  - Copy the smaller values into sorted array
  - Advance the pointer that pointed at smaller value

Example

Sort the array

```
26 13 1 14 15 38 2 27
```

1. Recap: Sorting Algorithms
2. Analysis of Mergesort
3. Quicksort
4. A General Lower Bound for Sorting

Analysis of Mergesort

Assumption

For simplicity: Array size N is a power of 2

Analysis of Mergesort

Assumption

For simplicity: Array size N is a power of 2

Runtime $T(N)$

$$\begin{aligned} T(1) &= 1 \\ T(N) &= 2T(N/2) + N \end{aligned}$$

Runtime of Mergesort

Theorem

$$T(N) = O(N \log N)$$

Runtime of Mergesort

Theorem

$$T(N) = O(N \log N)$$

Two Proof Techniques

- Telescoping a sum
- Continuous substitution

1. Recap: Sorting Algorithms
2. Analysis of Mergesort
3. Quicksort
   - Idea
   - Implementation
   - Analysis of Quicksort
4. A General Lower Bound for Sorting

Quicksort: Idea

Idea of Mergesort

Split array in two (trivial); sort the two (recursively); merge (linear)

Quicksort: Idea

Idea of Mergesort

Split array in two (trivial); sort the two (recursively); merge (linear)

Idea of Quicksort

Split array in two (linear); sort the two (recursively); merge (trivial)

Quicksort: Idea

Idea of Mergesort

Split array in two (trivial); sort the two (recursively); merge (linear)

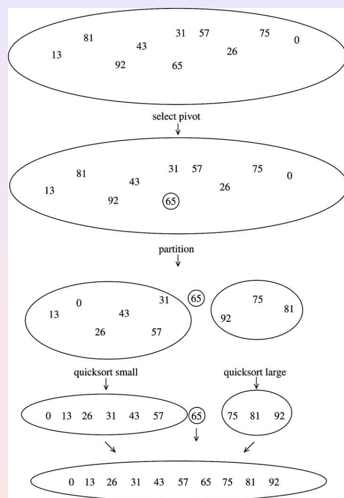
Idea of Quicksort

Split array in two (linear); sort the two (recursively); merge (trivial)

Splitting in Quicksort

For merging to be trivial, splitting is done around a *pivot* element. The first array contains the elements smaller than pivot; the second the elements larger than pivot

Example



Questionable Choice of Pivot

- Often, the pivot is chosen to be the first element

Questionable Choice of Pivot

- Often, the pivot is chosen to be the first element
- This is only ok if the input array is random

Questionable Choice of Pivot

- Often, the pivot is chosen to be the first element
- This is only ok if the input array is random
- If the input array has some order (almost sorted, or almost sorted in reverse), the pivot needs to be chosen differently

Common Choice of Pivot

A safe choice

Choosing a random element as pivot is a good choice in all cases.

Common Choice of Pivot

A safe choice

Choosing a random element as pivot is a good choice in all cases.
Random number generation is a bit of an overkill for choosing a pivot.

Common Choice of Pivot

A safe choice

Choosing a random element as pivot is a good choice in all cases.
Random number generation is a bit of an overkill for choosing a pivot.

Median-of-three

A choice that works well in practice is to choose the median of the first, last and middle elements

Implementation: Driver

```
public static <AnyType extends
                 Comparable<? super AnyType>>
void quicksort( AnyType [ ] a ) {
  quicksort( a, 0, a.length - 1 );
}
```

Implementation: Median-of-three Partitioning

```
private static <AnyType extends Comparable<? super
AnyType median3(AnyType [] a,int left,int right) {
  int center = ( left + right ) / 2;
  if( a[ center ].compareTo( a[ left ] ) < 0 )
    swapReferences( a, left, center );
  if( a[ right ].compareTo( a[ left ] ) < 0 )
    swapReferences( a, left, right );
  if( a[ right ].compareTo( a[ center ] ) < 0 )
    swapReferences( a, center, right );
  swapReferences( a, center, right - 1 );
  return a[ right - 1 ];
}
```

Implementation: Main Routine

```
void quicksort(AnyType [] a,int left,int right) {
  if( left + CUTOFF <= right ) {
    AnyType pivot = median3( a, left, right );
    int i = left, j = right - 1;
    for( ; ; ) {
      while( a[ ++i ].compareTo( pivot ) < 0 ) { }
      while( a[ --j ].compareTo( pivot ) > 0 ) { }
      if( i < j ) swapReferences( a, i, j );
      else break; }
    swapReferences( a, i, right - 1 );
    quicksort( a, left, i - 1 );
    quicksort( a, i + 1, right );
  } else insertionSort(a, left, right); }
```

Runtime of Quicksort

Definition

Let $i = |S_i|$ be the number of elements in the first partition

Runtime of Quicksort

Definition

Let $i = |S_i|$ be the number of elements in the first partition

Basic Quicksort relation

$$T(N) = T(i) + T(N - i - 1) + cN$$

Worst-case Analysis

The worst case

occurs when the pivot is always the smallest (or largest) element

Worst-case Analysis

The worst case

occurs when the pivot is always the smallest (or largest) element

Uneven split

$$T(N) = T(N-1) + 0 + cN$$

Worst-case Analysis

Adding up

$$T(N) = T(N-1) + 0 + cN = T(1) + c\sum_{i=2}^{N} i = O(N^2)$$

Best-case Analysis

The best case

occurs when the pivot is always the element exactly in the middle; both partitions have equal size

Best-case Analysis

The best case

occurs when the pivot is always the element exactly in the middle; both partitions have equal size

Even split

$$T(N) = 2T(N/2) + cN$$

Best-case Analysis

The best case

occurs when the pivot is always the element exactly in the middle; both partitions have equal size

Even split

$$T(N) = 2T(N/2) + cN$$

Similar to Mergesort

$$T(N) = O(N \log N)$$

Average Case of Quicksort

Similar to best case

$$T(N) = O(N \log N)$$

1. Recap: Sorting Algorithms
2. Analysis of Mergesort
3. Quicksort
4. A General Lower Bound for Sorting

Decision Trees

Root of decision tree

Root is initial state of algorithm

Decision Trees

Root of decision tree

Root is initial state of algorithm

Edges of decision tree

An edge represents the outcome of a comparison

Decision Trees

Root of decision tree

Root is initial state of algorithm

Edges of decision tree

An edge represents the outcome of a comparison

Leaves

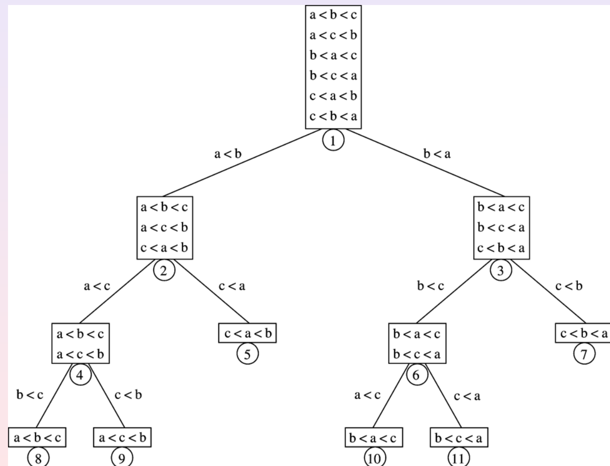
Each path to a leaf represents a run of the algorithm

Worst-Case Runtime

Worst-case in terms of tree depth

The number of comparisons required in the worst case is the depth of the deepest leaf

Example



Some Useful Facts

Leaves in binary tree

Let T be a binary tree of depth d. Then T has at most 2^d leaves.

Some Useful Facts

Leaves in binary tree

Let T be a binary tree of depth d. Then T has at most 2^d leaves.

Depth of a binary tree

A binary tree with L leaves must have depth at least $\lceil \log L \rceil$.

Some Useful Facts

Leaves in binary tree

Let T be a binary tree of depth d. Then T has at most 2^d leaves.

Depth of a binary tree

A binary tree with L leaves must have depth at least $\lceil \log L \rceil$.

Comparisons for sorting

Any sorting algorithm that uses only comparisons between elements requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case.

Main Theorem

Theorem

Any sorting algorithm that uses only comparisons between elements requires $\Omega(N \log N)$ comparisons.

Main Theorem

Theorem

Any sorting algorithm that uses only comparisons between elements requires $\Omega(N \log N)$ comparisons.

Proof

Prove that $\log(N!) \geq \Omega(N \log N)$.

Considerations for Choosing Sorting Algo

- Size of array: If array is very small (for example smaller than 20 or 30 elements), insertion sort is suitable

Considerations for Choosing Sorting Algo

- Size of array: If array is very small (for example smaller than 20 or 30 elements), insertion sort is suitable
- Cost of memory: Mergesort requires extra array; other algos don't

Considerations for Choosing Sorting Algo

- Size of array: If array is very small (for example smaller than 20 or 30 elements), insertion sort is suitable
- Cost of memory: Mergesort requires extra array; other algos don't
- Cost of comparisons: Comparator and Comparable require method calls for comparison, whereas comparisons are often inlined in C++

Considerations for Choosing Sorting Algo

- Size of array: If array is very small (for example smaller than 20 or 30 elements), insertion sort is suitable
- Cost of memory: Mergesort requires extra array; other algos don't
- Cost of comparisons: Comparator and Comparable require method calls for comparison, whereas comparisons are often inlined in C++
  Mergesort is known to have lowest number of comparisons among the popular sorting algorithms

Considerations for Choosing Sorting Algo

- Size of array: If array is very small (for example smaller than 20 or 30 elements), insertion sort is suitable
- Cost of memory: Mergesort requires extra array; other algos don't
- Cost of comparisons: Comparator and Comparable require method calls for comparison, whereas comparisons are often inlined in C++
  Mergesort is known to have lowest number of comparisons among the popular sorting algorithms
- Cost of moving elements: In Java, only references are changed; in C++ often objects themselves need to be moved

Next Week

- Friday: Midterm 2
- Topics: search, heaps, hashing, sorting