15-211
Fundamental Structures
of Computer Science

Introduction to Sorting

Ananda Guna
February 20, 2003

Announcements

- **Homework #4 is available**
  - Due on Monday, March 17, 11:59pm
  - *Get started now!*
- **Quiz #2**
  - Available on Tuesday, Feb.25
  - Some questions will be easier if you have some parts of HW4 working
- **Read**
  - Chapter 8

History

History of sorting from Knuth's book:

Hollerith's sorting machine developed in 1901-1904 used **radix sort**.

Card sorter: electrical connection with vat of mercury made through holes. First the 0s pop out, then 1s, etc.

For 2-column numerical data would sort of units column first, then re-insert into machine and sort by tens column.

History Ctd...

From NIST web site:

Hollerith's system-including punch, tabulator, and sorter-allowed the official 1890 population count to be tallied in six months, and in another two years all the census data was completed and defined; the cost was $5 million below the forecasts and saved more than two years‘time.

His later machines mechanized the card-feeding process, added numbers, and sorted cards, in addition to merely counting data.

In 1896 Hollerith founded the Tabulating Machine Company, forerunner of Computer Tabulating Recording Company (CTR). He served as a consulting engineer with CTR until retiring in 1921.

In 1924 CTR changed its name to IBM- the **International Business Machines** Corporation.

Comparison-based sorting

We assume

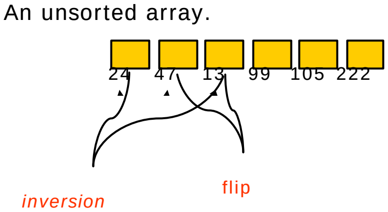
- Items are stored in an array.
- Can be moved around in the array.
- Can **compare** any two array elements.

Comparison has 3 possible outcomes:

<   =   >

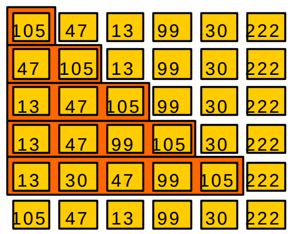
Flips and inversions

An unsorted array.



Two elements are inverted if A[i] > A[j] for i < j

Insertion sort



Insertion sort

```
for  i = 1  to  n-1  do
    insert a[i] in the proper place
        in  a[0:i-1]
```

```
So sub-array A[0..k-1] is sorted
for k = 1,.,n  after k-1 steps
```

Proof using loop invariants

```
for  i = 1  to  n-1  do {
  Invariant 1: A[0..i-1] is a sorted permutation of the original A[1..i-1]
  j = i-1; key = A[i];
  while (j >= 0 && A[j]>key)
    {  Invariant 2: A[j .. i-1] are all larger than the  key
       A[j+1] = A[j--] ;
    }
  A[j] = key;
}
```

- Proof is left as an exercise. Argue the correctness of the algorithm by proving that the loop invariants hold and then draw conclusions from what this implies upon termination of the loops.

Analysis of Insertion Sort

- In the i^{th} step we do at least 1 comparison, at most (i-1) comparisons and on average i/2 (call this C_i)
- M_i – the number of moves at the i^{th} step is $C_i + 2$
- Obtain formulas for C_{min}, C_{ave}, C_{max} and same for M_{min}, M_{ave}, M_{max}
- Exercise
- When is the worst case true? Best case true? What type of data sets?

How fast is insertion sort?

•Each step of the insertion sort we are reducing the number of inversions.

•Takes O(#inversions + N) steps, which is very fast if array is nearly sorted to begin with. I.e no inversions.

•We can slightly increase the performance by doing binary insertion

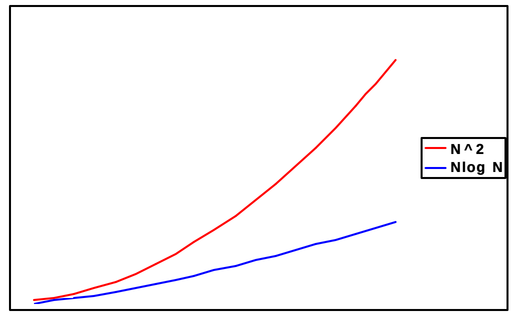
How long does it take to sort?

- Can we do better than $O(n^2)$?
  - In the worst case?
  - In the average case

Heapsort

- Remember *heaps*:
  - buildHeap has O(n) worst-case running time. That is $\Sigma\ 2^i(h\text{-}i) = O(n)$
  - deleteMin has O(log n) worst-case running time. So *n* deleteMin's would give a sorted list.
- Heapsort :
  - Build heap. O(n)
  - DeleteMin until empty. O(n log n)
  - *Total worst case*: O(n log n)

N^2 vs Nlog N



Sorting in O(n log n)

- Heapsort establishes the fact that sorting can be accomplished in O(n log n) worst-case running time.
- *In fact, later we will see that it is possible to prove that any sorting algorithm will have require at least O(n log n) in the worst case.*

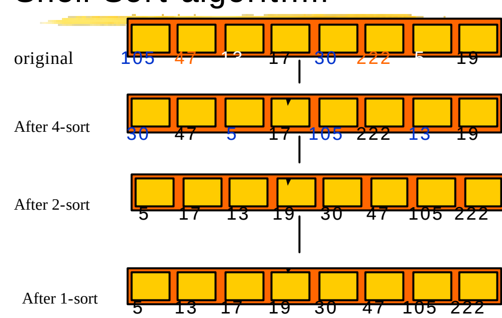
Heapsort in practice

- The average-case analysis for heapsort is somewhat complex.
- In practice, heapsort consistently tends to use nearly *n* log *n* comparisons. What if the array is sorted? What is the performance?
- So, while the worst case is better than n^2, other algorithms sometimes work better.

Shellsort

- ⌘ A refinement of insertion sort proposed by D.L.Shell in 1959
- ⌘Define k-sort as a process that sorts items that are k positions apart.
- ⌘So one can do a series of k-sorts to achieve a relatively few movements of data.
- ⌘A 1-sort is really the insertion sort. But then most items are in place.

Shell Sort algorithm



Shell Sort Analysis

- Each pass benefit from previous
  - Each i-sort combines two groups sorted in previous 2i-sort.
- Any sequence of increments (h_1,h_2,...) are fine as long as last one is 1.
  - $h_t = 1$, $h_{i+1} < h_i$
  - Each h-sort is an insertion sort
- Very difficult mathematical analysis

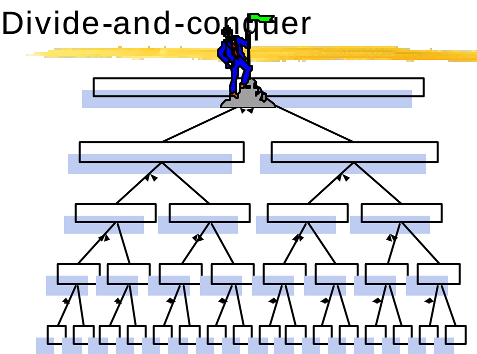
Shell Sort Analysis ctd..

- It is shown that for the sequence 1,3,7,15,31,... given by
  - h_t=1, h_{k-1}=2h_k+1 and t = logn –1
  - For this sequence, Shell Sort Algorithm is $O(n^{1.2})$
  - Proof available but difficult. Ignore till 15-451.

Recursive sorting

- If array is length 1, then done.
- If array is length N>1, then split in half and sort each half.
  - Then combine the results.

Divide-and-conquer



Divide-and-conquer is big

- *We will see several examples of divide-and-conquer in this course.*

Analysis of recursive sorting

- Let T(n) be the time required to sort n elements.
- Suppose also it takes time n to combine the two sorted arrays.
- Then:

Recurrence relation

- Then such “recursive sorting” is characterized by the following recurrence relation:
  - $T(1) = 1$
  - $T(n) = 2\,T(n/2) + n$

A solution

- A solution for
  - $T(1) = 1$
  - $T(N) = 2T(N/2) + N$
- is given by
  - $T(N) = N\log N + N$
  - which is $O(N\log N)$.
- *How to solve such equations?*

Exact solutions

- It is sometimes possible to derive closed-form solutions to recurrence relations.
- Several methods exist for doing this.

Repeated substitution method

- One technique is to use repeated substitution.
  - $T(N) = 2T(N/2) + N$
  - $2T(N/2) = 2(2T(N/4) + N/2)$
  - $= 4T(N/4) + N$
  - $T(N) = 4T(N/4) + 2N$
  - $4T(N/4) = 4(2T(N/8) + N/4)$
  - $= 8T(N/8) + N$
  - $T(N) = 8T(N/8) + 3N$
  - $T(N) = 2^kT(N/2^k) + kN$

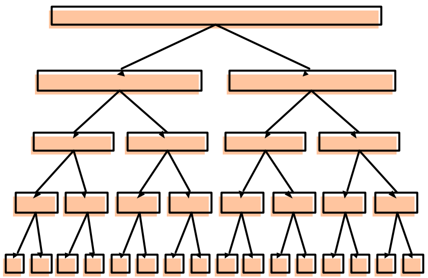
Repeated substitution, cont’d

- We end up with
  - $T(N) = 2^kT(N/2^k) + kN$, for all $k>1$
- Let’s use $k=\log N$.
  - Note that $2^{\log N} = N$.
- So:
  - $T(N) = NT(1) + N\log N$
  - $= N\log N + N$

Mergesort

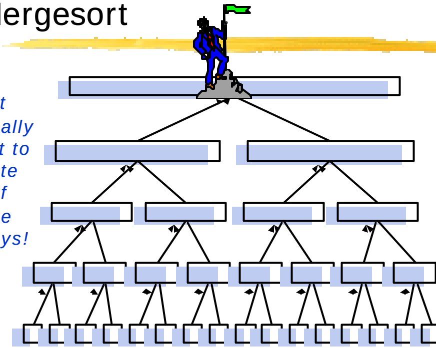
- Mergesort is the most basic recursive sorting algorithm.
  - Divide array in halves A and B.
  - Recursively mergesort each half.
  - Combine A and B by successively looking at the first elements of A and B and moving the smaller one to the result array.
- Note: Should be a careful to avoid creating of lots of result arrays.

Mergesort

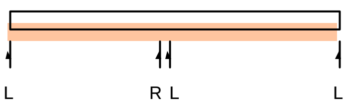


Mergesort

But don't actually want to create all of these arrays!



Mergesort



Use simple indexes to perform the split.

Use a single extra array to hold each intermediate result.

Analysis of mergesort

- Mergesort generates almost exactly the same recurrence relations shown before.
  - T(1) = 1
  - T(N) = 2T(N/2) + N - 1, for N>1
- Thus, mergesort is O(Nlog N).

Upper bounds for rec. relations

- Divide-and-conquer algorithms are very useful in practice.
- Furthermore, they all tend to generate similar recurrence relations.
- As a result, approximate upper-bound solutions are well-known for recurrence relations derived from divide-and-conquer algorithms.

Divide-and-Conquer Theorem

- Theorem: Let a, b, c $\geq$ 0.
- The recurrence relation
  - T(1) = b
  - T(N) = aT(N/c) + bN
  - for any N which is a power of c

a=2, b=1, c=2 for rec. sorting

- has upper-bound solutions
  - T(N) = O(N) if a<c
  - T(N) = O(Nlog N) if a=c
  - T(N) = $O(N^{\log_c a})$ if a>c

Upper-bounds

- Corollary:
- Dividing a problem into *p* pieces, each of size N/*p*, using only a linear amount of work, results in an *O*(Nlog N) algorithm.

Upper-bounds

- *Proof of this theorem later in the semester.*

Checking a solution

- It is also useful sometimes to check that a solution is valid.
  - This can be done by induction.

Checking a solution

- Base case:
  - T(1) = 1log 1 + 1 = 1
- Inductive case:
  - Assume T(M) = Mlog M + M, all M<N.
  - T(N) = 2T(N/2) + N

Checking a solution

- Base case:
  - T(1) = 1log 1 + 1 = 1
- Inductive case:
  - Assume T(M) = Mlog M + M, all M<N.
  - T(N) = 2T(N/2) + N

Checking a solution

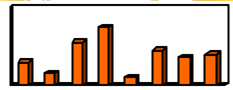
- Base case:
  - T(1) = 1log 1 + 1 = 1
- Inductive case:
  - Assume T(M) = Mlog M + M, all M<N.
  - T(N) = 2T(N/2) + N
  - = 2((N/2)(log(N/2))+N/2)+N
  - = N(log N - log 2)+2N
  - = Nlog N - N + 2N
  - = Nlog N + N

Quicksort

- Quicksort was invented in 1960 by Tony Hoare.
- Although it has $O(N^2)$ worst-case performance, on average it is $O(N \log N)$.
- *More importantly, it is the fastest known comparison-based sorting algorithm in practice.*

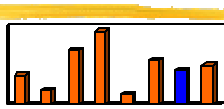
Quicksort idea

- Choose a pivot.

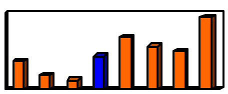


Quicksort idea

- Choose a pivot.

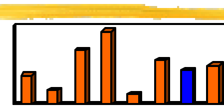


- Rearrange so that pivot is in the "right" spot.

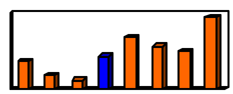


Quicksort idea

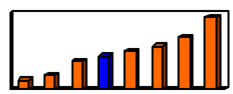
- Choose a pivot.



- Rearrange so that pivot is in the "right" spot.



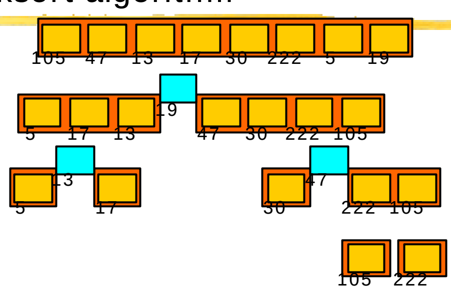
- Recurse on each half and conquer!



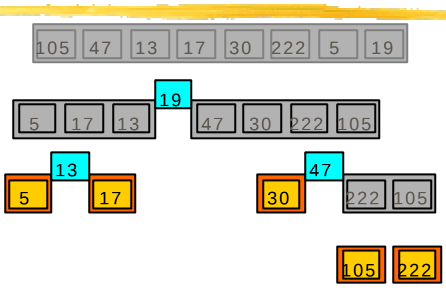
Quicksort algorithm

- If array A has 1 (or 0) elements, then done.
- Choose a *pivot element* x from A.
- Divide A-{x} into two arrays:
  - $B = \{y \in A \mid y \leq x\}$
  - $C = \{y \in A \mid y \geq x\}$
- Quicksort arrays B and C.
- Result is B+{x}+C.

Quicksort algorithm

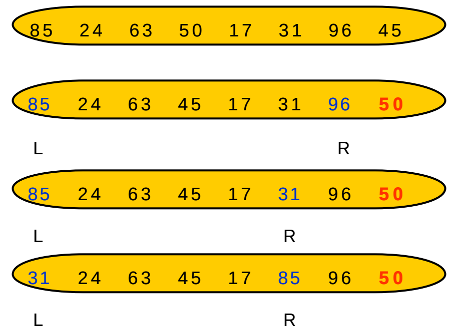


Quicksort algorithm

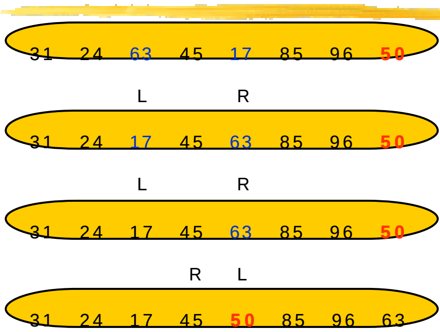


In practice, insertion sort is used once the arrays get "small enough".

Doing quicksort in place



Doing quicksort in place



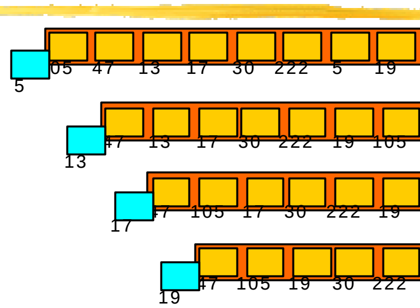
Quicksort is fast but hard to do

- Quicksort, in the early 1960's, was famous for being incorrectly implemented many times.
  - More about invariants next time.
- Quicksort is very fast in practice.
  - Faster than mergesort because Quicksort can be done "in place".

Informal analysis

- If there are duplicate elements, then algorithm does not specify which subarray B or C should get them.
  - Ideally, split down the middle.
- Also, not specified how to choose the pivot.
  - Ideally, the median value of the array, but this would be expensive to compute.
- *As a result, it is possible that Quicksort will show $O(N^2)$ behavior.*

Worst-case behavior



Analysis of quicksort

- Assume random pivot.
  - T(0) = 1
  - T(1) = 1
  - T(N) = T(i) + T(N-i-1) + cN, for N>1
    - where I is the size of the left subarray.

Worst-case analysis

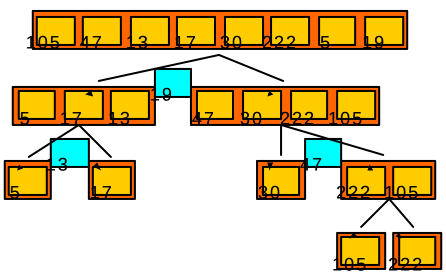
- If the pivot is always the smallest element, then:
  - T(0) = 1
  - T(1) = 1
  - T(N) = T(0) + T(N-1) + cN, for N>1
  - $\cong$ T(N-1) + cN
  - = $O(N^2)$
- *See the book for details on this solution.*

Best-case analysis

- In the best case, the pivot is always the median element.
- In that case, the splits are always "down the middle".
- Hence, same behavior as mergesort.
- That is, $O(N\log N)$.

Average-case analysis

- Consider the quicksort tree:



Average-case analysis

- The time spent at each level of the tree is $O(N)$.
- So, on average, how many levels?
  - That is, what is the *expected height* of the tree?
  - If on average there are O(log N) levels, then quicksort is O(Nlog N) on average.

Summary of quicksort

- A fast sorting algorithm in practice.
- Can be implemented in-place.
- But is $O(N^2)$ in the worst case.
- Average-case performance?