Divide-and-Conquer

1. Divide the problem into subproblems
2. Conquer subproblems by solving recursively (unless small enough)
3. Combine/Stitch solutions to subproblems

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Mergesort

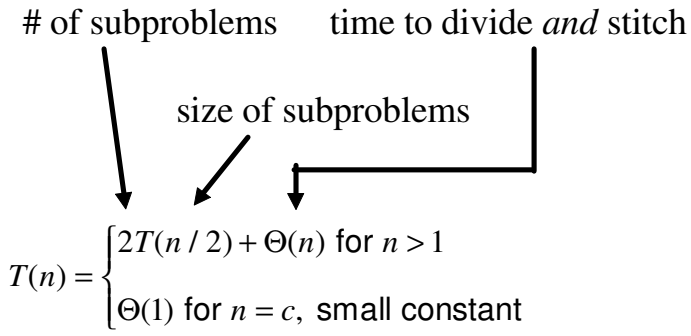
Divide: Divide the n-element sequence to be sorted into 2 subsequences of $n/2$ items each.

Conquer: Recursively sort each of the 2 subsequences.

Stitch: Combine the two sorted sequences into the sorted result.

This represents a "top-down" description of the divide-and-conquer approach.

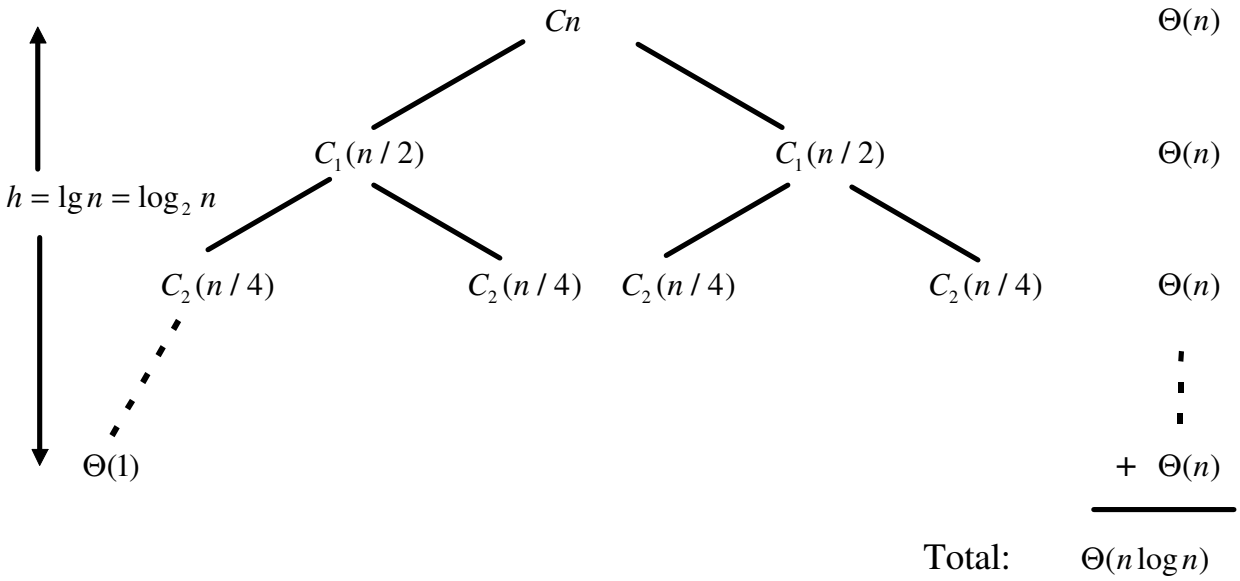
Discuss mergesort w.r.t. linked lists (sequentially). Discuss the running time as:



Using the *Master Method* (CLR, chapter 4) $\Rightarrow T(n) = \Theta(n \log n)$.
Remind students to go and review the *master method* from either CLR or Sahni or some discrete math books.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Recursion Tree for MergeSort



Master Method (CLR, pg. 62):

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we interpret *n/b* to be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large *n*, then $T(n) = \Theta(f(n))$.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Mergesort on a Linear Array

1. Reminder: we already have an optimal tractor-tread (rotation)-based algorithm for the linear array.
2. Discuss (simple) straightforward implementation of mergesort on an array based on single rotation for each of the log n phases.
   a) First rotation to find position of each element in other list that will be merged into - count number of elements in front in current list and also in the list to merge into.
   b) Second rotation to move elements to proper positions in new list.
   c) This algorithm yields a running time of $\Theta(n \log n)$, which is suboptimal.
3. Discuss modifications so that the rotations are restricted to the appropriate regions.
   a) Go over the running time: $T(n) = T(n/2) + \Theta(n)$, which is $\Theta(n)$.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Bitonic Sort on Hypercube (Review)

Given N data items per processor on a hypercube of size 2^d (i.e., $N2^d$ total items), discuss sorting on a *medium-grained* machine with bitonic sort. Show how to integrate the basic fine-grained algorithm for message passing with efficient sequential algorithms for local merging. The running time is

$$\Theta\left(N \log N + \frac{d(d+1)}{2} N\right),$$

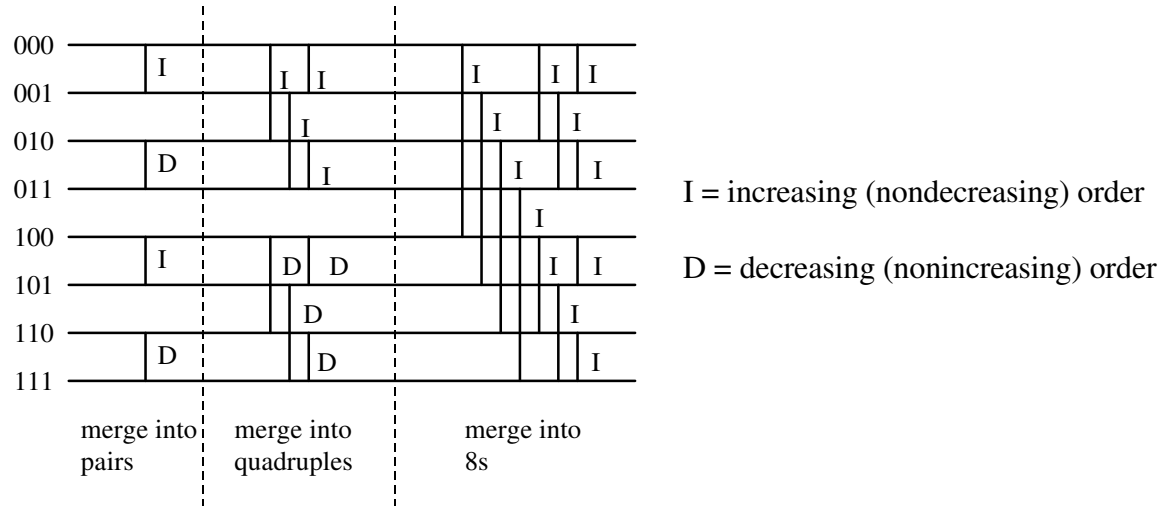
where the $N \log N$ term represents the initial serial sorting time, while the $\frac{d(d+1)}{2} N$ term is the parallel time necessary to perform the exchanging and merging.

<Remainder of this page is from an earlier set of notes - just here for review purposes - don't need to go over all details, but the diagram at the bottom may be of help in serving as a reminder>

Sorting: Build bottom-up using bitonic merge units for merging a bitonic sequence into a sorted list. To sort 2n elements, initially presented in an arbitrary order, requires $\Theta(\log^2 n)$ levels and $\Theta(n \log^2 n)$ comparators.

So, sorting $= \sum_{i=1}^{\log_2 2n} i = \frac{(\log_2 2n)(\log_2 2n+1)}{2}$ levels $= \frac{\log^2(2n)}{2} + \frac{\log(2n)}{2}$

Another view: Sorting 8 random values into nondecreasing order.



Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Bitonic Sort on a Mesh

Theorem: Given n items arbitrarily distributed one per processor on a MCC of size n, they can be sorted into any predefined (and reasonably computable) order in $\Theta(n^{1/2})$ time.

We demonstrate how to sort into *row-major* order:

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Example:

10	9	14	2	9	10	14	2	9	4	14	12
4	15	11	12	15	4	11	12	15	10	11	2
6	1	5	13	1	6	13	5	1	3	13	7
8	3	7	0	8	3	0	7	8	6	0	5

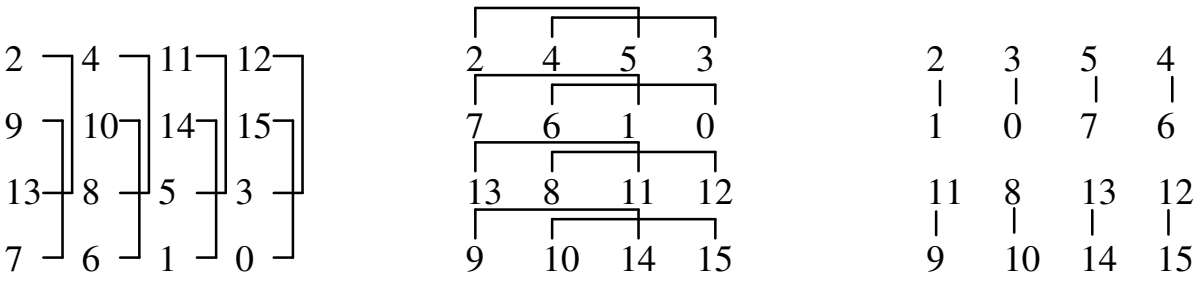
initially ordered — Now sorted into — into 1×1 sections — 1×2 sections

4	9	14	12	4	9	14	12	4	2	11	12
10	15	11	2	10	2	11	15	10	9	14	15
1	3	13	7	13	7	1	3	13	8	5	3
6	8	5	0	6	8	5	0	6	7	1	0

Now sorted into

2×2 sections

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.



Now sorted into

2×4 sections

1 — 0 5 — 4
2 — 3 7 — 6
9 — 8 13 — 12
11 — 10 14 — 15

0 1 4 5
2 3 6 7
8 9 12 13
10 11 14 15

Now sorted into
4×4 section

1. Discuss sorting into any predetermined order in $\Theta(n^{1/2})$ time.
2. Discuss relationship between sorting and RAR/RAW.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Concurrent Read/Write

Concurrent Read: A concurrent read may be used in a situation where a set of processors wishes to obtain data associated with a set of keys, but where there is no a priori knowledge as to which processor maintains the data associate with any particular key.

For example, processor P_i might need to know the data associated with the key `blue,' but might not know which processor in the system is responsible for maintaining the information associated with the key "blue." In fact, all processors in the system might be requesting one or more pieces of data associated with, not necessarily distinct, keys.

Concurrent Write: A concurrent write may be used in a situation where a set of processors wishes to update the data associated with a set of keys, but again do not necessarily know for any key, which processor is responsible for maintaining the data associated with that key.

Discuss some examples.

These **concurrent read/write** operations generalize the read/write operations of a PRAM by making them *associative*, i.e., locating data by key rather than by address.

In order to maintain consistency during concurrent read and concurrent write operations, it will be assumed that there is at most one *master record*, stored in some processor, associated with each unique key. In a concurrent read, every processor generates one *request record* corresponding to each key that it wishes to receive information about (a small fixed number). A concurrent read allows multiple processors to request information about the same key. A processor requesting information about a nonexistent key will receive a null message at the end of the operation.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Implementation: One implementation of a ***concurrent read*** on a parallel machine with *n* processors, follows (show examples to go along with the steps that follow):

1. Every processor creates C_1 *master records* of the form (Key, Return Address, data, "MASTER"), where C_1 is the maximum number of keyed master records maintained by any processor, and Return Address is the index of the processor that is creating the record. (Processors maintaining less than C_1 master records will create dummy records so that all processors create the same number of master records.)
2. Every processor creates C_2 *request records* of the form (Key, Return Address, data, "REQUEST"), where C_2 is the maximum number of request records generated by any processor, and Return Address is the index of the processor that is creating the record. (Processors requesting information associated with less than C_2 master records will create dummy records so that all processors create the same number of request records.) Notice that the data fields of the request records are presently undefined.
3. *Sort* all $(C_1 + C_2)n$ records together by the Key field. In case of ties, place records with the flag "MASTER" before records with the flag "REQUEST."'
4. Use a *broadcast* within ordered intervals to propagate the data associated with each master record to the request records with the same Key field. This allows all request records to find and store their required data.
5. Return all records to their original processors by *sorting* all records on the Return Address field.

Therefore, the time to perform a concurrent read, as described, is bounded by the time to perform a fixed number of sort and interval operations.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Concurrent Write (overview):

- For the **concurrent write** it will be assumed that there is at most one *master record*, stored in some processor, associated with each unique key.
- Processors generate *update records* which specify the key and piece of information about that key that they wish to update.
- If two or more update records contain the same key, then a master record will be updated with the minimum data value of these records. (In other circumstances, one could replace minimum with any other commutative, associative, binary operation.)
- The concurrent write may be accomplished by a method similar to that of the concurrent read, in which case the time to complete the operation is bounded by the time required for a fixed number of sort and interval operations.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Quicksort (Partition Sort)

Linked List:

1. Explain the generic *linked list* implementation.
2. Discuss this as being the *natural* approach as per C.A.R. Hoare
3. Wave hands at O(n log n) running time, but do not go over details until we have dealt with the array implementation.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Array Implementation:

1. Traditionally presented for historic reasons - linked lists did not exist many years ago.
2. Presented because analysis is interesting.
3. Presented because algorithmic issues are interesting in terms of extensions and augmentations.
4. Also, there are still some well used dusty-deck routines that involve sorting and for which arrays were/are the only choice of a major data structure.
5. However, this is **NOT THE NATURAL/INTUITIVE IMPLEMENTATION OF QUICKSORT**, regardless of what many authors may claim/imply.

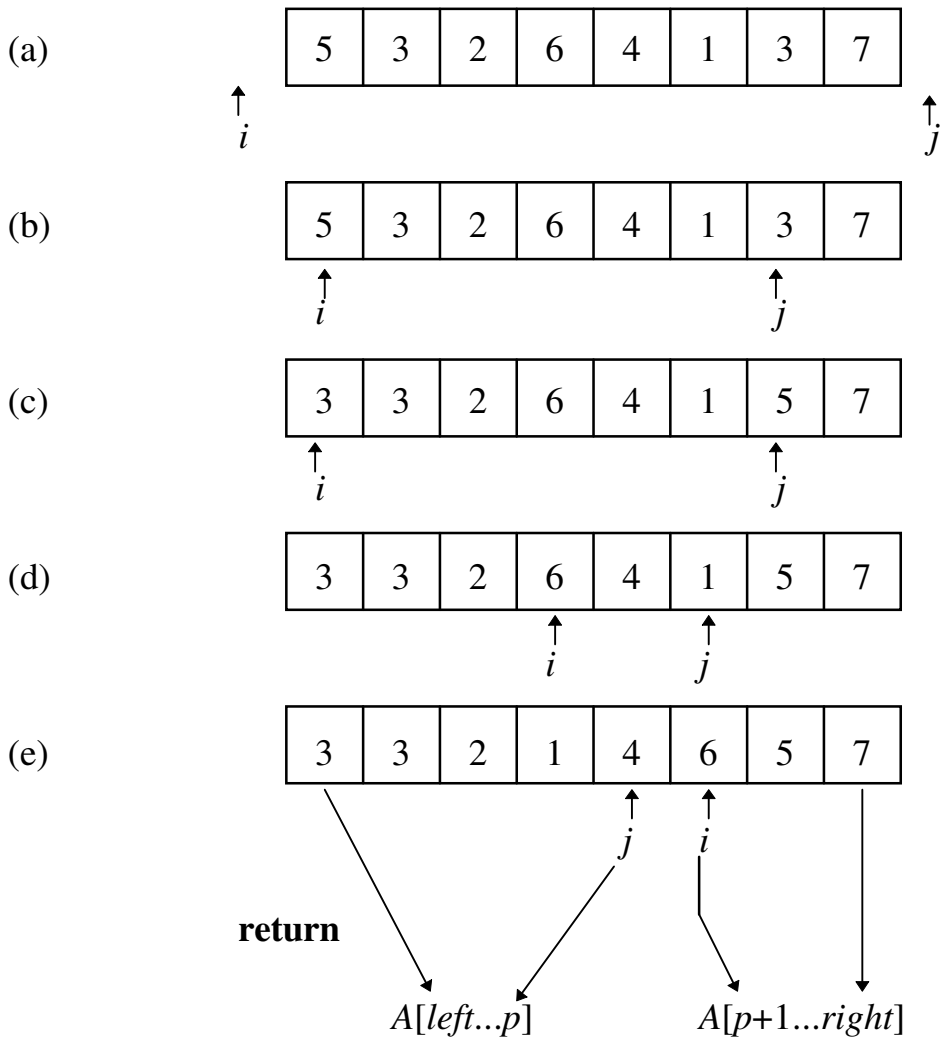
```
Quicksort (A,left,right)
if left < right then    p ← Partition(A,left,right)
                        Quicksort(A,left,p)
                        Quicksort(A,p+1,right)
end if
```

1. **Divide**: $A[left...right]$ is partitioned into 2 **nonempty** subarrays $A[left...p]$ and $A[p+1...right]$ such that all elements in $A[left...p] \leq A[p+1...right]$.
2. **Conquer**: Sort the two subarrays, $A[left...p]$ and $A[p+1...right]$, recursively.
3. **Stitch:** Requires no work since in an array.

```
Partition(A,left,right)
x ← A[left]
i ← left - 1
j ← right + 1
while True do
      repeat j ← j - 1 until A[j] ≤ x
      repeat i ← i +1 until A[i] ≥ x
      if i < j then SWAP(A[i],A[j])
      else return j
end while
```

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Example:



Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Analysis of Quicksort

Discuss the following in detail. Spend a bunch of time on this:

Time:	Best-case	$\Theta(n \log n)$	evenly divided
	*Worst-case	$\Theta(n^2)$	partition into 1 and n-1
	Expected-case	$\Theta(n \log n)$	we will prove this (?)
Space:	Best-case	$\Theta(1)$	only if $\Theta(n^2)$ running time
	Worst-case	$\Theta(n)$	only if $\Theta(n^2)$ running time
	Expected	$\Theta(\log n)$	when $\Theta(n \log n)$ running time

* - This happens in the case of ordered data, which is a fairly frequent situation. Notice that for ordered data, insertion sort is $\Theta(n)$ time.
In general, though, worst-case quicksort is still as fast as either insertion sort or selection sort.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Expected-Case Analysis of Quicksort

< This proof is probably not worth doing based on years and years of observing the cost-benefit analysis in terms of the students education.>

Assume distinct keys randomly distributed.

- Let $k(i)$ be the expected number of key comparisons required to sort *i* items.
- Note that $k(0) = k(1) = 0$; $k(2) = 3$.

Partition Assumptions:

1. To sort $A[1...N]$, use $A[1]$ as the partition element.
2. After partitioning, if $A[1]$ is the i^{th} element, it can be placed in position *i*. We assume this modification as it aids in the analysis.
3. Therefore, the recursive calls are performed on $A[1...(i-1)]$ and $A[(i+1)...N]$.

Partition: number of comparisons

1. It takes (*N*+1) comparisons to partition the *N* elements (2 where *i* and *j* meet).
2. On average, it takes $k(i-1)$ and $k(n-i)$ comparisons to sort each subarray, resp.

Probability of position of split:

Since it is equally likely that the partition element chosen will wind up anywhere from position $1...N$ $\Rightarrow$ The probability is 1/*N* that the split will be at any of the *N* positions.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

So, $k(n) = \frac{1}{n}\sum_{i=1}^{n}[k(i-1) + k(n-i) + (n+1)]$, where

- $k(n)$ is the expected number of key comparisons,
- $\frac{1}{n}$ is the expected case analysis,
- $k(i-1)$ and $k(n-i)$ are the expected number of key comparisons for the recursive calls, and
- (n+1) is the time required for the partition.

$$k(n) = \frac{1}{n}\sum_{i=1}^{n}[k(i-1) + k(n-i) + (n+1)] \quad = \quad (n+1) + \frac{1}{n}\sum_{i=1}^{n}[k(i-1) + k(n-i)]$$

$$= \quad (n+1) + \frac{1}{n}\begin{bmatrix} k(0) + k(n-1) \\ +k(1) + k(n-2) \\ +\ldots\ldots\ldots+\ldots\ldots\ldots \\ +k(n-1) + k(0) \end{bmatrix}$$

$$= \quad (n+1) + \frac{1}{n}2\sum_{i=1}^{n-1}k(i) \text{ , note that } k(0)=0$$

$$k(n) \; = \quad (n+1) + \frac{2}{n}[k(n-1) + k(n-2) + k(n-3)+\ldots+k(1)]$$

Define $S = k(n-2) + k(n-3)+\ldots+k(1)$

Then $$k(n-1) = (n-1) + 1 + \frac{2}{n-1}[k(n-2) + k(n-3)+\ldots+k(1)]$$

That is, $$k(n-1) = (n-1) + 1 + \frac{2}{n-1}[S]$$

or $$k(n-1) = n + \frac{2}{n-1}S$$

Therefore, $$S = \frac{n-1}{2}(k(n-1) - n)$$

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

So,
$$k(n) = n+1+\frac{2}{n}\left[k(n-1)+\frac{n-1}{2}\left(k(n-1)-n\right)\right]$$
$$= n+1+\frac{2}{n}k(n-1)+\frac{n-1}{n}\left(k(n-1)-n\right)$$
$$= n+1+\frac{2}{n}k(n-1)+\frac{n-1}{n}k(n-1)-(n-1)$$
$$= 2+k(n-1)\left(\frac{n+1}{n}\right)$$

Notice that $k(n)/(n+1) = 2/(n+1) + k(n-1)/n$

Let $X(n) = k(n)/(n+1) \Rightarrow k(n-1)/n = X(n-1)$

So,

$$X(n) = \frac{2}{n+1}+X(n-1) = \frac{2}{n+1}+\frac{2}{n}+X(n-2) = \frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+X(n-3) = \ldots\ldots$$
$$= \frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+\ldots+\frac{2}{4}+X(2)$$
$$= 2\left(\frac{1}{4}+\frac{1}{5}+\ldots+\frac{1}{n+1}\right)+C,\ C = X(2) \text{ a constant } = k(2)/3 = 3/3 = 1$$
$$= C+2\sum_{i=4}^{n+1}\frac{1}{i} \le C+2\sum_{i=1}^{n+1}\frac{1}{i}$$
$$= C+2H_{n+1}$$
$$\approx C+2\log(n+1)$$
$$= \Theta(\log n)$$

So, $k(n) = (n+1)X(n) = \Theta(n\log n)$ expected-case number of comparisons.

{Proof of expected-case number of data moves is similar.}

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Improving Quicksort

Improving the *running time*:

1. Pick better middle key: e.g., choose the median of 3 keys
   a) For large data sets (e.g., > 1000 keys) this gives 5% improvement
   b) Improves expected-time and reduces likelihood of worst-case performance.
2. Randomize the data to reduce the possibility of worst-case performance.
3. Quicksort is fast for large n, but slow for small n (e.g., $n<10$). Therefore, build a **hybrid** algorithm:
   a) Pick m. Sort as long as $right - left \geq m$.
   b) Use **insertion sort** to clean up: Run it exactly once at the end of the algorithm where it is now known that every element is no more than m positions from its final destination. Therefore, this final cleanup step requires $O(mn)$ time.

Improving the *space*:

1. Eliminate recursion and keep a stack.
2. Do the two consecutive recursive calls by pushing them onto the stack and not pushing the current process onto the stack.
3. Always push the larger piece onto the stack and *then* push the smaller piece onto the stack. In this way, the smaller piece is evaluated first. Therefore, there will be no more than $\lfloor \log_2 n \rfloor,\ n > 1,$ items on the stack at any time. This gets rid of the $\Theta(n)$ worst-case space problem.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.

Mention Bruce Wagar's extension to Quicksort, termed Hyperquicksort, for medium- and coarse-grained models. Discuss different options and the effect they have on the running time of the algorithms.

Hyperquicksort:

1. Distribute the items evenly to all nodes (if necessary). Each of the 2^d nodes now has N items.
2. Each node sorts its items using a $\Theta(N \log N)$ time algorithm.
3. Node 0 picks its median item, call it A, and broadcasts its key to all nodes.
4. Each node partitions its local set of data into 2 groups: those $\leq A$ and those $> A$.
5. Break up the entire cube of size 2^d into two subcubes, each of size 2^{d-1}, referred to as **L** and **U**. Each node in **L** sends its items $> A$ to its adjacent node in **R**. Likewise, nodes in **U** send their items $\leq A$ to their adjacent nodes in **L**. Notice that when this step is complete, all items $\leq A$ are in **L**, while all those $> A$ are in **U**.
6. Each node now merges together the set of data just received with the one it kept so that the items in the node are once again sorted. Important to note that is only requires a *merge* operation of the two ordered data sets.
7. Repeat steps 3-6 on each of **L** and **U** simultaneously, recursively, and in parallel until the subcubes consist of a single node, at which point the data in the entire hypercube will be sorted.

The expected-case running time (assumes that the data starts out in a random fashion) is

$$\Theta\left(N \log N + \frac{d(d+1)}{2} + dN \right)$$

where the $N \log N$ term represents the sequential running time, the $\frac{d(d+1)}{2}$ term represents the broadcasting from step 3, and the dN term represents the time required for the exchanging and merging. Given good splits, this represents an improvement of $\frac{(d+1)}{2}$ in the running time over the medium-grained version of bitonic sort.

Note that items will not necessarily wind up evenly distributed like bitonic sort.

Copyright © 1996-1997 by Russ Miller. All Rights Reserved.