CSE 326
Lecture 20

Spring Quarter 1998

Sorting Algorithms

- Insertion sort - slow for unsorted keys, but useful for sorting nearly sorted keys.
- Quicksort - in-place sort with very good average performance, needs a stack or recursion.
- Mergesort - very good worst case performance, need an auxiliary array for merging.
- Heapsort - in-place sort with good worst case performance, needs no auxiliary storage.
- Radix sort - very good worst case performance especially for long keys, needs auxiliary storage. Is not a comparison sort.
- Other sorts: Shellsort, selection sort, bubble sort, bucket sort may be useful in some settings.

Internal Sorting

- Input: an array A[0..n-1] of unsorted keys (integers for example) with pointers to records
- Goal: sort the array according to the keys.

12	20	13	30	5	19	9	17	25	10	6	22

↓ sort

5	6	9	10	12	13	17	19	20	22	25	30

Insertion sort

12	20	13	30	5	19	9	17	25	10	6	22
12	20	13	30	5	19	9	17	25	10	6	22
12	20	13	30	5	19	9	17	25	10	6	22
12	13	20	30	5	19	9	17	25	10	6	22
12	13	20	30	5	19	9	17	25	10	6	22
5	12	13	20	30	19	9	17	25	10	6	22
5	12	13	19	20	30	9	17	25	10	6	22

Insertion Sort

- for i = 1 to n-1 do insert(A,i)
- Invariant: after insert(A,i), the subarray A[0..i] is sorted
- Properties:

  + in-place
  + simple and elegant
  + fast for small data sets < 10
  + good for nearly sorted data

  - $O(n^2)$ time worst case and on average

Quicksort Partitioning

19	20	13	30	5	12	9	17	25	10	6	22

pivot i swap j

19	6	13	30	5	12	9	17	25	10	20	22

i j

19	6	13	30	5	12	9	17	25	10	20	22

i swap j

19	6	13	10	5	12	9	17	25	30	20	22

i j

19	6	13	10	5	12	9	17	25	30	20	22

j i

17	6	13	10	5	12	9	19	25	30	20	22

Quicksort using Median of 3

0	1	2	3	4	5	6	7	8	9	10	11
22	20	13	30	5	19	9	17	25	10	6	12

sort A[0] , A[5], A[11]

12	20	13	30	5	19	9	17	25	10	6	22

pivot = 19, swap A[0] and A[5]

19	20	13	30	5	12	9	17	25	10	6	22

partition A[0..11]

17	6	13	10	5	12	9	19	25	30	20	22

sort A[0] , A[3], A[6]

9	6	13	10	5	12	17	19	25	30	20	22

Quicksort using Median of 3

0	1	2	3	4	5	6	7	8	9	10	11
9	6	13	10	5	12	17	19	25	30	20	22

pivot = 10, swap A[0] and A[3]

10	6	13	9	5	12	17	19	25	30	20	22

partition A[0..6]

9	6	5	10	13	12	17	19	25	30	20	22

sort A[0] , A[1], A[2]

5	6	9	10	13	12	17	19	25	30	20	22

sort A[4] , A[5], A[6]

5	6	9	10	13	12	17	19	25	30	20	22

Quicksort using Median of Three

- quicksort(l,r: integer) // sorts the subarray A[l,r]

```
pivot local key; k local integer;
  if r < l + c then sort A[r..l] by a small set sorting method
  else
     sort A[l], A[(r+l)/2], A[r]
     pivot = A[(l+r)/2];
     swap A[l] and A[(l+r)/2]; // pivot is now in position l of A
     k := partition(A,l,r,pivot); // pivot belongs in position k of A
     swap A[l] and A[k];
     quicksort(l, k-1);
     quicksort(k+1, r);
end
```

- Call quicksort(0,n-1) to accomplish the entire sort.

Notes on Quicksort

+ Very fast on average O(n log n)

+ Worst case O(n^2) but very unlikely using median of three

+ Good cache performance because subproblems eventually fit in the cache

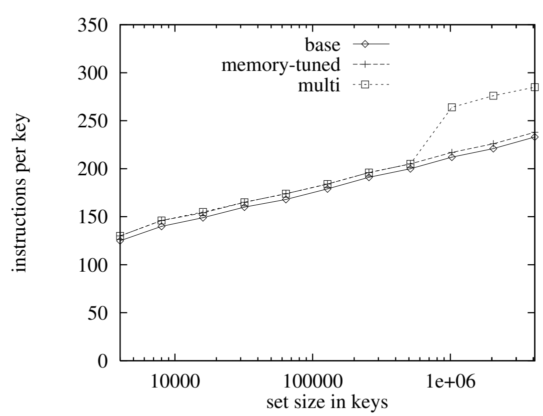
+ In-place sort

- Worst case O(n^2)

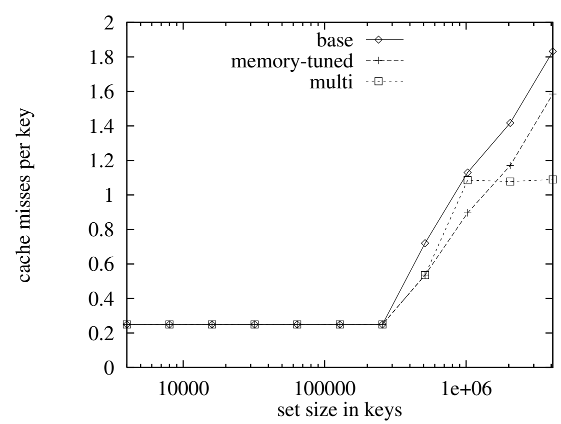
- Uses extra storage for the recursion stack

Weiss and others suggest not sorting the small subsets immediately when encountered. Quicksort will then leave the array slightly unsorted. One pass of insertion sort will work sort the nearly sorted arrays. Because of cache performance, this may be a bad idea for large data sets.

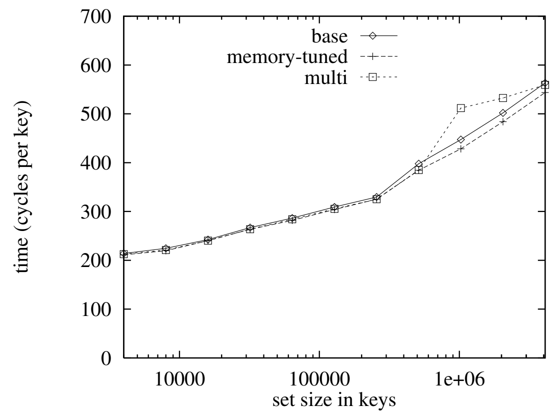
Quicksort Instruction Count



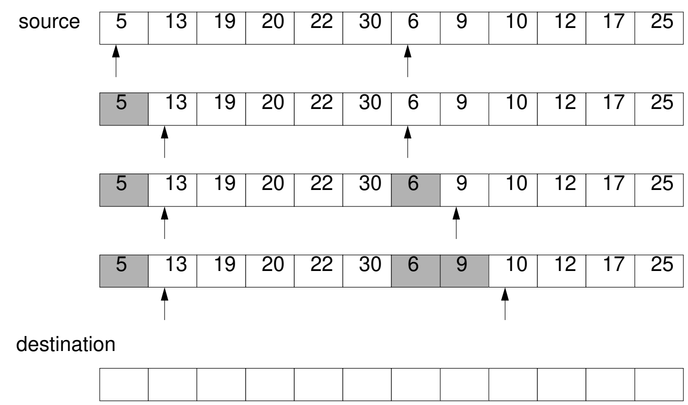
Quicksort Cache Misses



Quicksort Execution Time



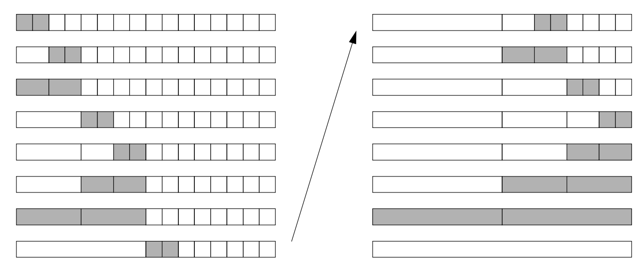
Merging in Mergesort



Recursive Mergesort

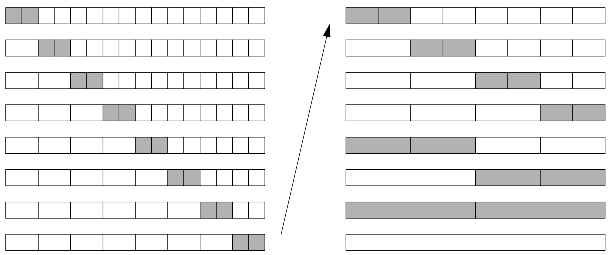
- mergesort(A: array of keys, i,j: integer)
  B local array of keys; k local integer;
  if i < j then
    k:= (i+j)/2;
    mergesort(A,i,k);
    mergesort(A, k+1,j);
    merge A[i..k] with A[k+1..j] into B[i..j];
    copy B[i..j] into A[i..j]
  end

Merging Steps in Recursive Mergesort



Iterative Mergesort

- Merge pairs of sets of size 1
  Merge pairs of sets of size 2
  Merge pairs of sets of size 4
  etc.



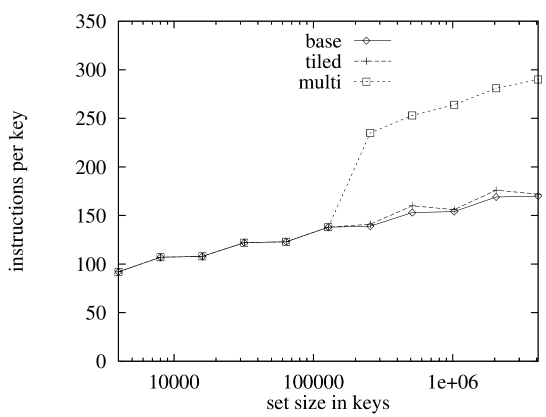
Mergesort Notes

- Copying can be essentially eliminated by alternating the merging passes from one array to another. A final copy back may be needed if the number of merge passes is odd.
- Standard iterative mergesort can have poor cache performance, but good organization can solve the cache miss problem
- Properties of mergesort

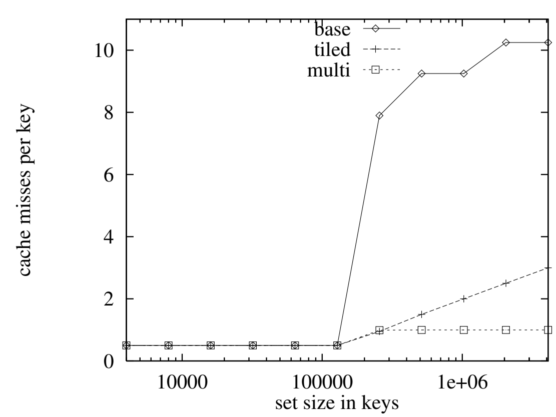
  + Very fast - competitive with quicksort
  + O(n log n) worst case time
  + Good cache performance can be achieved by using recursive mergesort or with tricks with iterative mergesort

  - Uses an auxiliary array

Mergesort Instruction Count



Mergesort Cache Misses



Mergesort Execution Time