COMP2521 19T0

Week 6, Thursday: Order! Order (II)

Jashank Jeremy

jashank.jeremy@unsw.edu.au

more sorting algorithms
non-comparing sorts

Sorting

Divide-and-Conquer Algorithms

divide-and-conquer algorithms
break up, or shard, the problem
into (easier) computations on smaller pieces,
and combine the results.

(usually) easy to implement recursively!
(usually) easy to implement in parallel!

Divide-And-Conquer Sorting

1. If a collection has less than two elements, it's sorted. Otherwise, split it into 2 parts.
2. Sort both parts separately.
3. Combine the sorted collections to return the final result.

Sorted Merge

Copy elements from the inputs one at a time,
giving preference to the smaller of the two.
When one list is empty,
copy the rest of the elements from the other.

```
/// In-place merge on array `a',
/// of slices `a[lo..mid-1]' and `a[mid..hi]'.
void merge (Item a[], size_t lo, size_t mid, size_t hi);

/// Out-of-place merge, into `dest[0..ndest-1]',
/// from `a[0..nA-1]' and `b[0..nB-1]'.
void merge (Item dest[], size_t ndest,
        Item a[], size_t nA, Item b[], size_t nB);
```

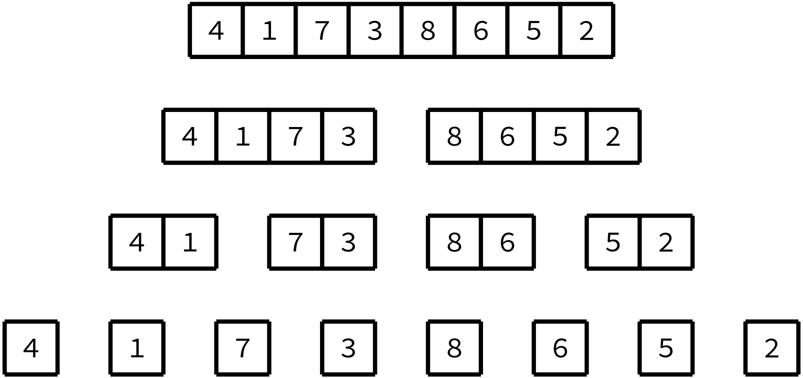
Merge Sort

A divide-and-conquer sort:

partition the input into two equal-sized parts.
recursively sort each of the partitions.
merge the two now-sorted partitions back together.

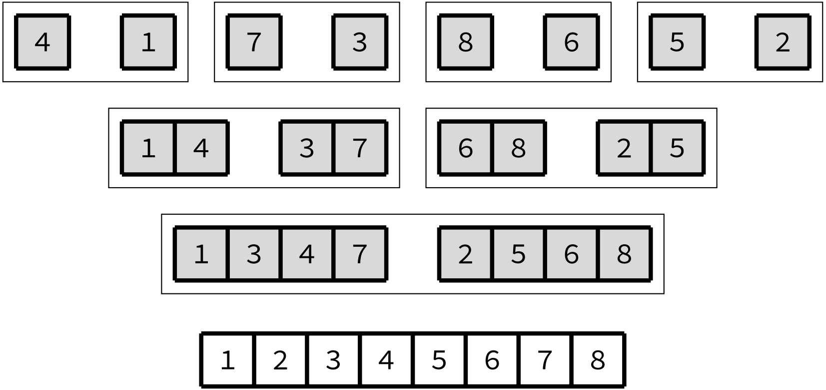
Merge Sort

Demo (I)



Merge Sort

Demo (II)



Merge Sort

C Implementation: Sort

```
void sort_merge (Item a[], size_t lo, size_t hi)
{
    if (hi <= lo) return;
    size_t mid = (lo + hi) / 2;
    sort_merge (a, lo,     mid);
    sort_merge (a, mid+1, hi);
    merge (a, lo, mid, hi);
}
```

Merge Sort

C Implementation: Merge

```
void merge (Item a[], size_t lo, size_t mid, size_t hi)
{
    Item *tmp = calloc (hi - lo + 1, sizeof (Item));
    size_t i = lo, j = mid + 1, k = 0;

    // Scan both segments, copying to `tmp'.
    while (i <= mid && j <= hi)
        tmp[k++] = less (a[i], a[j]) ? a[i++] : a[j++];

    // Copy items from unfinished segment.
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi)  tmp[k++] = a[j++];

    // Copy `tmp' back to main array.
    for (i = lo, k = 0; i <= hi; a[i++] = tmp[k++]);

    free (tmp);
}
```

Merge Sort

Analysis (I)

How many steps does it take to sort a collection of N elements?

Splitting arrays into two halves: constant time.
To re-combine, N steps.

$$T\left(N\right) = N + 2\,T\left(N/2\right)$$

substitute $N := 2^N$; then:

$$T\left(2^N\right) = 2^N + 2\,T\left(2^N/2\right)$$
$$T\left(2^N\right) = 2^N + 2\,T\left(2^{N-1}\right)$$

Merge Sort

Analysis (II)

divide out 2^N; then:

$$T\left(2^N\right)/2^N = 1 + 2\,T\left(2^{N-1}\right)/\left(2^N\right)$$
$$T\left(2^N\right)/2^N = 1 + T\left(2^{N-1}\right)/\left(2^{N-1}\right)$$

expanding, we get:

$$1 + \left(1 + T\left(2^{N-2}\right)/\left(2^{N-2}\right)\right)$$
$$1 + \left(1 + \left(1 + T\left(2^{N-3}\right)/\left(2^{N-3}\right)\right)\right)$$
$$\ldots = N$$

$$T\left(2^N\right)/2^N = N$$
$$T\left(2^N\right) = 2^N N$$

$$T\left(N\right) = N\log_2 N$$

Merge Sort

Analysis (III)

How many steps does it take
to sort a collection of N elements?

- split array into equal-sized partitions
  halving at each level $\Rightarrow \log_2 N$ levels
- same operations happen at every recursive level
- each 'level' requires $\leq N$ comparisons —
  worst case: two arrays exactly interleaved, N comparisons

Merge Sort

Analysis; Summary

Merge sort is $O(n \log n)$.

Generally, stable...
... as long as the merge is stable.

Not in-place:
$O(n)$ memory for merge; $O(\log n)$ stack space.

Oblivious:
$O(n \log n)$ best case, average case, worst case

Merge Sort on Lists

Straightforward!

- Traverses input in sequential order.
- Don't need extra space for merging list.
- Works top-down and … bottom-up?

Bottom-Up Merge Sort

An approach that works non-recursively!

- on each pass, our array contains sorted *runs* of length m.
- initially, N sorted runs of length 1.
- The first pass merges adjacent elements into runs of length 2.
- The second pass merges adjacent elements into runs of length 4.
- … continue until we have a single sorted run of length N.

Can be used for *external* sorting;
e.g., sorting disk-file contents

Bottom-Up Merge Sort

C Implementation

```
#define MIN(a,b) ((a) < (b) ? (a) : (b))

void sort_merge_bu (Item a[], size_t lo, size_t hi)
{
    for (size_t m = 1; m <= lo - hi; m *= 2)
        for (size_t i = lo; i <= hi - m; i += 2 * m) {
            size_t end = MIN (i + 2*m - 1, hi);
            merge (a, i, i + m - 1, end);
        }
}
```

Merge Sort is Slow

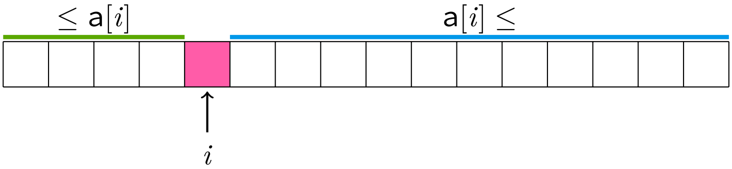
Merge sort uses a trivial split operation;
all the heavy lifting is in the *merge* operation.

Can we split the collection in a more intelligent way,
so combining the results is easier?

…*e.g.*, making sure all elements in one part
are less than elements in the second part?

Better Partitioning

to partition array a
at some index i (the ‘pivot’),
we need to swap elements such that,
for other indices j and k,
$j < i$ implies $\mathsf{a}[j] \leq \mathsf{a}[i]$
$k > i$ implies $\mathsf{a}[i] \leq \mathsf{a}[k]$



Quick Sort

C Implementation: Sort

Assuming we have a partition function,
this looks very similar to merge sort.

```
void sort_quick_naive (Item a[], size_t lo, size_t hi)
{
    if (hi <= lo) return;
    size_t part = partition (a, lo, hi);
    sort_quick_naive (a, lo, part ? (part - 1) : 0);
    sort_quick_naive (a, part + 1, hi);
    // look, ma! no merge!
}
```

Quick Sort

C Implementation: Naïve Partition

```c
size_t partition (Item a[], size_t lo, size_t hi)
{
    Item v = a[lo]; // our `pivot' value.
    size_t i = lo + 1, j = hi;
    for (;;) {
        while (less (a[i], v) && i < j) i++;
        while (less (v, a[j]) && i < j) j--;
        if (i == j) break;
        swap_idx (a, i, j);
    }
    j = less (a[i], v) ? i : i - 1;
    swap_idx (a, lo, j);
    return j;
}
```

Quick Sort

Analysis

How many steps does it take
to sort a collection of N elements?

N steps to partition an array...
constant-time combination of sub-results.

best-case (equal sized partitions): $O(N \log N)$
worst-case (one part contains all elements):
$T(N) = N + T(N-1) = N + (N-1) + T(N-2)$
... $= N(N+1)/2$, which is $O\left(N^2\right)$

Quick Sort

Summary

Quick sort with naïve partition is...

Unstable (in this implementation)...
... but can be made stable.

In-place: partitioning is done in-place;
stack depth is $O(N)$ worst-case, $O(\log N)$ average

Oblivious.

Problems with Quick Sort

Picking the first or last element as pivot
is an absolutely terrible life choice.

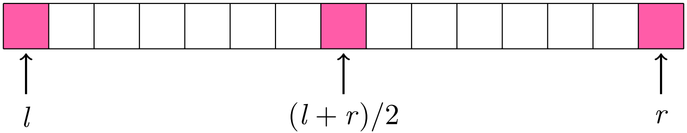
... existing order is a worst case.
... existing *reverse* order is a worst case.
`partition` always gives us parts of size $N - 1$ and 0.

Our ideal pivot is the *median* value.
Our worst pivot is the largest/smallest value.
We can reduce the probability of picking a bad pivot...

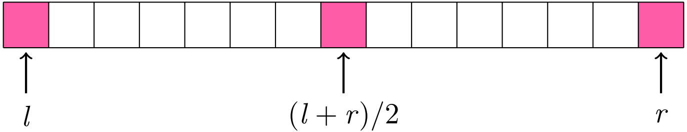
Quick Sort with Median-of-Three Partition

Pick three values: left-most, middle, right-most.
Pick the median of these three values as our pivot.

Ordered data is no longer a worst-case scenario.
In general, doesn't eliminate the worst-case ...
... but makes it much less likely.



Quick Sort with Median-of-Three Partitioning



1. Pick a$[l]$, a$[r]$, a$[(l + r)/2]$
2. Swap a$[r - 1]$ and a$[(l + r)/2]$
3. Sort a$[l]$, a$[r - 1]$, a$[r]$, such that a$[l] \leq$ a$[r - 1] \leq$ a$[r]$
4. Partition on a$[l + 1]$ to a$[r - 1]$.

Quick Sort with Median-of-Three Partitioning

C Implementation

```
void qs_median3 (Item a[], size_t lo, size_t hi)
{
    size_t mid = (lo + hi) / 2;
    if (less (a[mid], a[lo])) swap_idx (a, lo, mid);
    if (less (a[hi], a[mid])) swap_idx (a, mid, hi);
    if (less (a[mid], a[lo])) swap_idx (a, lo, mid);
    // now, we have a[lo] <= a[mid] <= a[hi]
    // swap a[mid] to a[lo+1] to use as pivot
    swap_idx (a, lo+1, mid);
}

void sort_quick_m3 (Item a[], size_t lo, size_t hi)
{
    if (hi <= lo) return;
    qs_median3 (a, lo, hi);
    size_t part = partition (a, lo + 1, hi - 1);
    sort_quick_m3 (a, lo, part ? (part - 1) : 0);
    sort_quick_m3 (a, part + 1, hi);
}
```

Quick Sort Optimisations

Sub-file Cutoff

For small sequences (when $n < 5$, say),
quick sort is expensive
because of the recursion overhead.

With a sub-file cutoff, we have two choices:
use a different algorithm on small partitions; or
do a second sort after the quicksort finishes.
(Insertion sort is a good choice: lots of almost-sorted data!)

Quick Sort Optimisations

Bentley-McIlroy's Three-Way Partition

For sequences with many duplicate keys, partitioning can screw up badly.

instead, do a three-way partition:
keys $< a[i]$, $= a[i]$, $> a[i]$.

Quick Sort on Lists

Straightforward to do…
if we just use the first or last element as pivot
(which means we're vulnerable to ordered data again)

using a random or median-of-three pivot
is now $O(n)$ not $O(1)$

Quick Sort vs Merge Sort

Design of modern CPUs mean,
for sorting arrays in RAM
quicksort *generally* outperforms mergesort.

quicksort is more ‘cache friendly’:
good locality of access on arrays

on the other hand, mergesort is
readily stable, readily parallel,
more efficient with slower data;
a good choice for sorting linked lists

The $n \log n$ Lower Bound

How Low Can We Go? (I)

If we have 3 items,
then $3! = 6$ possible permutations as input.
(n items implies $n!$ possible permutations.)

If we do 1 comparison,
we can form two categories (true, false).
(k comparisons implies 2^k categories.)

The $n \log n$ Lower Bound

How Low Can We Go? (I)

n items implies $n!$ possible permutations.
k comparisons implies 2^k categories.

We need to do enough comparisons so
$n! \leq 2^k$.
$log_2 n! \leq log_2 2^k$
$log_2 n! \leq k$
… applying Stirling's approximation, and waving our hands:
$n \log n < k$.

the theoretical lower bound on
worst-case execution time
for comparison-based sorts is $O(n \log n)$.
(Quicksort, mergesort are pretty much
as good as it gets, for unknown data.)

Non-Comparison-Based Sorting

All the sorts so far have been
comparison-based sorts.
(They compare things, using some ordering relation $\leq$.)
Works on *any* data, so long as we have $\leq$.

What if we know more about the keys?
… could we get down to $O(n)$ time?

Key-Indexed Counting Sort

count up the number of times each key appears;
this indexes where each item belongs in the sorted array

FOR EXAMPLE:
assuming my key domain is numbers $[0 \ldots 10]$,
if we have three '0's, and two '1's,
'2's must go at index 5 and onwards

look, ma! no comparisons!
look, ma! **an $O(n)$ sort!**
terms and conditions apply, see in store for details

Key-Indexed Counting Sort

we must know our sequence is of size N,
and the domain of keys in that sequence.

pumped-up KICS pretty efficient
... if M is small compared to N.
actually, $O(n + M)$ ... so if we have $1, 2, 999999$ ...

Not in-place — uses a temporary array.
Can be stable! Not really adaptive.

Heap Sort

We already have a data structure
which has element ordering as an invariant:
the heap or priority queue.

We *could* just dump all n elements
into a priority queue, and dequeue them —
n operations of $O(\log n)$ complexity.
no gain.

What if we used the heap-fix-down mechanism
on the whole array, popping off the maximum item,
and shrinking the heap each time? That's $O(n)$!
The catch: the inner loop is expensive.

Heap Sort

C Implementation

```
void sort_heap (Item a[], size_t lo, size_t hi)
{
    size_t N = hi - lo + 1;
    Item *pq = &a[lo - 1];
    for (size_t k = N/2; k >= 1; k--)
        heap_fixdown (pq, k, N);
    while (N > 1) {
        swap_idx (pq, 1, N);
        heap_fixdown (pq, 1, --N);
    }
}
```