Outline

Announcement: out of town Thu 11/8, Tue 11/13, Tue 11/20
There will be guest lectures: John Umbaugh and Chris Kiekentveld

No office hours Thu 11/8, Fri 11/9, Tue 11/13, Tue 11/20

Last time: Sorting
  ● Selection sort
  ● Heap sort
  ● Insertion sort

Today: More sorting
  ● Insertion sort optimizations
  ● Merge sort
Insertion Sort

Algorithm (see Fig. 15.2):

- given a “pile” of items in an array
- insert into an already sorted array one item at a time using linear search
- non-adaptive version: continues linear search till end of sorted array
- adaptive version: stops linear search once an insertion point has been found

Contrast to selection sort:

- selection sort: select item from pile in order, add to the front (end) of sorted list
- insertion sort: pick next item from pile, insert in order into sorted list
Insertion Sort

Sort array $a[5, 2, 3, 8, 5, 1]$:

Non-adaptive:

```
void isort_na(int *a, int N)
```

Is it stable?

Adaptive:

```
void isort_a(int *a, int N)
```

Is it stable?
Running Time and Optimizations

Running time:

“Optimizations”:

- binary search instead of linear search?
- move instead of swap: saves $O(n^2)$ copies
- use of sentinel: saves $O(n^2)$ CMPs
Insertion Sort: Move Instead of Swap

Slide items down instead of swapping them, to move down \( k \) steps, only need \( k + 2 \) copies instead of \( 3k \) copies

Sort array \( a[5, 2, 3, 8, 5, 1] \):

```c
void
isort_am(int *a, int N)
```

Saves \( O(n^2) \) copies, but total running time is still \( O(n^2) \)

Is it stable?

Sugih Jamin (jamin@eecs.umich.edu)
Insertion Sort: Use of Sentinel

Sort array \(a[5, 2, 3, 8, 5, 1]\):
Can use sentinel to do \(O(n^2)\) less comparisons

- move smallest item to the leftmost position \((O(n)\) operation)
- then it’s not necessary to check \((j > 0)\) in the inner loop (saves \(O(n^2)\) CMPs)

Total running time is still \(O(n^2)\)

```c
void isort_as(int *a, int N)
{
    int i, j, tmp;

    for (j = 1, i = j+1; i < N; i++) {
        if (a[j] > a[i]) {
            j = i;
        }
    } /* now move smallest to a[0] */
    if (a[0] > a[j]) swap(a[0], a[j]);

    for (i = 2; i < N; i++) {
        for (j = i, tmp=a[i];
            a[j-1] > tmp; j--) {
            a[j] = a[j-1];
        }
        a[j] = tmp;
    }
    return;
}
```

Sugih Jamin (jamin@eecs.umich.edu)
Inversion

A sorted list is but a permutation of an unsorted one

Let $S = \{1, 2, 3, \ldots, n\}$ and $P = \{p_1, p_2, \ldots, p_n\}$ be a permutation of $S$

An inversion in $P$ is a pair of items $(p_i, p_j)$ where $p_i > p_j$ but $i < j$,
e.g., $P = \{1, 4, 3, 2\}$ has 3 inversions: $(4, 3), (4, 2), (3, 2)$

The job of a sorting algorithm is to “correct” inversions!
Insertion Sort Average Running Time

Let $P^R$ be $P$ in reverse, i.e., $P^R = \{p_n, p_{n-1}, \ldots, p_2, p_1\}$

A pair of items $(p_i, p_j)$ is either an inversion in $P$ or in $P^R$, with equal probability

There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs in a sequence of $n$ items

On average, we expect $\frac{n(n-1)/2}{2} = \frac{n(n-1)}{4}$ inversions in a given sequence $P$

Insertion sort corrects one inversion at a time, so the average running time of insertion sort is $\frac{n(n-1)}{4} = O(n^2)$
Merge Sort

Sort $a[85 \ 24 \ 63 \ 45 \ 17 \ 31 \ 96 \ 50]$ using merge sort

$[85 \ 24 \ 63 \ 45 \ 17 \ 31 \ 96 \ 50]$
$[85 \ 24 \ 63 \ 45] \ [17 \ 31 \ 96 \ 50]$
$[85 \ 24] \ [63 \ 45] \ [17 \ 31] \ [96 \ 50]$
$[85] \ [24] \ [63] \ [45] \ [17] \ [31] \ [96] \ [50]$
$[24 \ 85] \ [45 \ 63] \ [17 \ 31] \ [50 \ 96]$
$[24 \ 45 \ 63 \ 85] \ [17 \ 31 \ 50 \ 96]$
$[17 \ 24 \ 31 \ 45 \ 50 \ 63 \ 85 \ 96]$
Merge Sort

Algorithm (see Preiss Fig. 15.10):

- split list into two roughly equal parts
- mergesort each part recursively
- merge the two sorted parts

```c
void mergesort(int *a, int left, right)
{
    int mid;

    if (left <= right) return;
    mid = (left+right)/2;
    mergesort(a, left, mid);
    mergesort(a, mid+1, right);
    merge(a, left, mid, right);
    return;
}
```
Merge

Splitting list into two roughly equal parts is easy

How to merge the two sorted parts?

Simplest way: use scratch space of equal size (see Preiss Fig. 15.9)

```c
void merge(int *a, left, mid, right)
```

Running time of merge:

Need an extra $O(N)$ space, which may not be available for large data set or small devices (handhelds, e.g.)
Merge Sort

Recurrence relation:

Running time:

Disadvantages:

• **Non-adaptive**: for pre-sorted sequence, worse than insertion sort!
• **Not in-place**: needs an extra $O(n)$ space, with attending copy operation, so slower than other $O(n \log n)$ sort
• **Stack space**: recursion uses up $O(\log n)$ stack space

Advantage: can be **stable** if `merge()` is stable
Removing Recursion: Iterative Merge Sort

Bottom-up:
- consider original list as $N$ sublists of size 1
- scan thru list performing 1-by-1 mergers to produce $N/2$ ordered sublists of 2 elements
- scan thru list performing 2-by-2 mergers to produce $N/4$ ordered sublists of 4 elements
- etc.

```c
void
imsort(int *a, left, right)
{
    int size, i, n;
    for (size = 1, n = right-left;
         size <= n; size *= 2) {
        for (i = left; i <= right-size;
             i += 2*size) {
            merge(a, i, i+size-1,
                 min(i+2*size-1,right));
        }
    }
    return;
}
```

Sugih Jamin (jamin@eecs.umich.edu)
Inefficiencies of `merge()`

From `merge(int *a, left, mid, right)`:

```c
1   for (i=k=left, j=mid+1; k <= right; k++) {
2       if (i > mid) { a[k] = tmp[j++]; continue; }
3       if (j > right) { a[k] = tmp[i++]; continue; }
5   }
```

The boundary checks in lines 2 and 3 return false most of the time. Can we remove them by using a **sentinel** as in insertion sort?

Sugih Jamin (jamin@eecs.umich.edu)
Insertion Sort: Use of Sentinel

Sort array $a[5, 2, 3, 8, 5, 1]$: Can use sentinel to do $O(n^2)$ less comparisons

- move smallest item to the leftmost position ($O(n)$ operation)
- then it’s not necessary to check ($j > 0$) in the inner loop (saves $O(n^2)$ CMPs)

Total running time is still $O(n^2)$

```c
void isort_as(int *a, int N)
{
    int i, j, tmp;

    for (j = 1, i = j+1; i < N; i++) {
        if (a[j] > a[i]) {
            j = i;
        }
    } /* now move smallest to a[0] */
    if (a[0] > a[j]) swap(a[0], a[j]);

    for (i = 2; i < N; i++) {
        for (j = i, tmp=a[i]; a[j-1] > tmp; j--) {
            a[j] = a[j-1];
        }
        a[j] = tmp;
    }
    return;
}
```

Sugih Jamin (jamin@eecs.umich.edu)
Merge Using Bitonic Sort

```c
void bsmerge(int *a, left, mid, right)
{
    int i, j, k, tmp[MAXN];

    memcpy(tmp, &a[left], mid-left+1);
    for (k = mid-left+1, j = right; j > mid; j--, k++)
        tmp[k] = a[j]; /* copy in reversed order */
    for (i=k=left, j=right; k <= right; k++) {
        a[k] = ( tmp[j] < tmp[i] ) ? tmp[j--] : tmp[i++];
    }
    return;
}
```

But `bsmerge` is not stable. Why?
In-place Merge Sort

Huang and Langston (1988) has a version of in-place merge (http://www.cs.utk.edu/ langston/courses/cs594-fall2003/HL.pdf) but it is very slow

Alternatively, Katajainen, Pasanen, and Teuhola (1996) does in-place mergesort directly:

- mergesort the first half of the array using the second half as scratch space, by swapping unsorted elements in the second half into the first half (make sure not to overwrite!)
- sorted half ended up in the latter half of array
- sort the first 1/4 using the next 1/4 as scratch space
- now the second 1/4 subarray and the latter 1/2 subarray are sorted, merge the two subarrays using the first 1/4 as scratch space
- repeat for the last, still unsorted 1/4 subarray

Performance: 50% slower than using temporary array

Sugih Jamin (jamin@eecs.umich.edu)