**Sorting**

**Algorithm:** Insertion Sort  
Input: unsorted list/array $U$

1. Initialize: $S = \emptyset$ (sorted)
2. while $U$ not empty
   (a) remove $x$ from $U$ (arbitrary)
   (b) insert $s$ into $S$ in sorted order.

Claim: selection/insertion sort is correct.  
Proof:
- invariant: $S$ is always sorted.
- induction on $|S|$.

**Algorithm:** Selection Sort  
Input: unsorted list/array $U$

1. Initialize: $S = \emptyset$ (sorted)
2. while $U$ not empty
   (a) select (and remove) minimum $x$ from $U$
   (b) append $x$ to $S$.

Claim: selection/insertion sort is $\Theta(n^2)$.  
Proof:
- in step $i$, insertion takes $\Theta(i)$;  
  total = $\sum_i i = \Theta(n^2)$
- in step $i$, selection takes $\Theta(n - i)$;  
  total = $\sum_i (n - i) = \Theta(n^2)$

**In-place sorting**

“Sort an array without any additional storage”

**Idea:** First part of array is sorted, second part unsorted.

**Algorithm:** In-place Selection Sort

```c
void selection_sort(int [] array, int n) {
    int i, j, min, temp;
    for (i=0; i<n; i++) {
        min = i;
        for (int j=i+1; j<n; j++)
            if (array[j] < array[min])
                min=j;
        temp = array[min];
        array[min] = array[i];
        array[i] = temp;
    }
}
```

**Note:**
• invariant: first $i$ elements of array are sorted!
• append $x$: add $x$ at position $i$, increment $i$.

Faster Sorting

Algorithm: Heap-sort

Input: unsorted list/array $U$

1. Initialize: $S = H = \text{build-heap}(U)$
2. while $H$ not empty
   (a) $x = \text{delete-min}(U)$.
   (b) append $x$ to $S$.

In-place Heap-sort

Recall:

```c
void build_max_heap(int *array,int n)
// Input: array[1..n]
// Output: array[1..n] is max-heap.

int delete_max(int *array, int n)
// Input: max-heap array[1..n]
// Output: max element,
// array[1..(n-1)] is max-heap.
```

Algorithm: In-place Heap-sort
Input: array[0..(n-1)]

```c
void selection_sort(int *array, int n)
{
    array--; // array[1]...array[n]
    build_max_heap(array,n);
    for (int i = n; i > 1; i--)
        array[i] = delete_max(array,i);
}
```

Claim: Heap-sort is correct.
Claim: Heap-sort runtime is $\Theta(n \log n)$.

Proof: build-heap is $\Theta(n)$; delete-min is $\Theta(\log n)$ ($n$ times); Total = $\Theta(n \log n)$.

Merge-sort

Algorithm: Merge-sort
Input: unsorted list $U$

0. if $|U| \leq 1$, return $U$.
1. partition $U$ into $U'$ and $U''$ (equal size)
2. $S' = \text{Merge-sort}(U')$ and $S'' = \text{Merge-sort}(U'')$.
3. return Merge($S',S''$).

Algorithm: Merge
Input: sorted lists $S, T$

0. if $S$ empty, return $T$ (and vice versa).
1. let $(x',S') = S$ and $(y',T') = T$
2. if $x' < y'$
    return $(x',\text{Merge}(S',T'))$.
3. else $(y' \leq x'$
    return $(y',\text{Merge}(S,T'))$.

Claim: Merge-sort is correct.

Proof: induction on $|U|$
Claim: Merge-sort runs in time $\Theta(n \log n)$.

Proof:

- let $T(n)$ be runtime on size $n$ input.
- Base Case: $T(1) = 1$.
- Recursive Case: $T(n) = 2T(n/2) + n$
• Solve for $T(n)$

\[
T(n) = T(n/2) + n
\]

\[
= 2(2T(n/4) + n/2) + n
\]

\[
= 4T(n/4) + n + n
\]

\[
= 4(2T(n/8) + n/4) + n + n
\]

\[
= 8T(n/8) + n + n + n
\]

\[
\vdots
\]

\[
= nT(1) + n + n + \cdots + n
\]

\[
= n \log n
\]

- before query, $k$ possible permutations
- after query, $\geq k/2$ possible permutations.

• best case: $\log(n!)$ queries
\[
(= \Omega(n \log n)).
\]

□

Lower Bounds

Claim: Any comparison-based sorting algorithm has $\Omega(n \log n)$ comparisons in worst-case.

Analogy:

- view as game between The Algorithm and Nature (a.k.a., The Adversary)
- Algorithm decides which elt’s to compare, Adversary chooses outcome of comparison.
- Note: The Adversary must be consistent (with some permutation).
- Any sorting algorithm eventually pins the adversary to a single permutation.

Proof: (Algorithm vs. Adversary)

- number of permutations $= n!$.
- for each query, Adversary chooses answer with most uncertainty:

Example:

- Consider $(a, b, c)$
- Many possible permutations: $a < b < c$ or $a < c < b$
  or $b < a < c$ . . .
- Algorithm queries: “$a < b$?”
- Adversary answers: “yes” or “no”.
- if “yes”, algorithm can eliminate permutations with $a > b$.
- if “no”, algorithm can eliminate permutations with $b \leq a$.
- Algorithm must figure out which of remaining permutations is correct.