## Sorting

- Ordering items in increasing order (of some key)
- Very useful in practice
- Part of various algorithms
- Page numbering in books
- Grading
- Google search
- $\ldots$

Most of the topics covered here are from chapters:
6.1 (insertion sort),
6.2 (quicksort),
6.3 (lower bound on sorting)
6.4 (counting and radix sort)
6.5 (selection)
9.1 (simple text search)
9.2 (Rabin-Karp Algorithm)

## Insertion Sort:

– Array $A[1 : n]$ given
– Progressively, insert the $i$-th element of the array into $A[1 : i-1]$, which is already sorted.
Example:

$5, 3, 7, 9, 2, 6$
$5, 3, 7, 9, 2, 6$
$3, 5, 7, 9, 2, 6$
$3, 5, 7, 9, 2, 6$
$3, 5, 7, 9, 2, 6$
$2, 3, 5, 7, 9, 6$
$2, 3, 5, 6, 7, 9$

Input: $A[1:n]$.
Output: Sorted array in increasing order.

Insertion Sort

```
1. For $i = 2$ to $n$ {
    2.  Let $temp = A[i]$
    3.  Let $j = i - 1$
    4.  While $j \geq 1$ and $A[j] > temp$ Do {
                $A[j+1] = A[j]$
                $j = j - 1$
        }
    5.  $A[j+1] = temp$.
    }
End
```

Correctness:

In the iteration of the For loop at step 1, for a particular value of $i$:
(a) just before the start of the iteration: $A[1], \dots, A[i-1]$ are in increasing order,
(b) during the iteration: $A[i]$ is placed in its correct position.
(c) at the end of the iteration: $A[1], \dots, A[i]$ are in increasing order.

(b): In each iteration of the For loop:
The while loop: "shifts" the numbers greater than $A[i]$ to the right.
Then places $A[i]$ in its correct place.

Complexity:
Number of comparisons/operations:
Best Case: $O(n)$
(happens when the numbers are already in increasing order).
The while loop condition is never true, and thus the algorithm takes a constant amount of time in each iteration of the for loop.

Worst Case:
For the iteration of the For-Loop for a particular value of $i$:
The while loop is executed at most $i - 1$ times. (* happens for example when the numbers are in reverse sorted order *)
Thus, the whole while loop takes time at most $c_1 * i$
Thus, the for loop iteration for a particular $i$ takes time $\leq c_2 * i$.
Therefore the whole algorithm takes time at most:
$c_2(2 + 3 + 4 + 5 \ldots + n) \leq c_2 * n^2$.

Another method to analyze:
$T(n) \leq T(n-1) + C * n$.
This gives, $T(n) = O(n^2)$.

- $T$(particular input) =?
- Size of input
- Worst Case $T(n)$, for a particular size $n$ of input:
  max { T(input): input has size $n$}
- Best Case $T(n)$
  min { T(input): input has size $n$}
- Average Case $T(n)$

$$\frac{\Sigma\{\text{ T(input): input has size } n\}}{\text{number of inputs of size } n}$$

- Sometimes Average Case is with probability over different inputs of size $n$.
- Asymptotic Bounds, are over "all $n$". Gives good estimate for large enough $n$, ...,

## Quicksort

- Choose a pivot element.

  $5, 2, 7, 5, 9, 4$

- partition the numbers into two parts, those $<$ pivot and those $\geq$ pivot.

  $4, 2, 5, 5, 9, 7$

- Sort each halves recursively.

Partition$(A, i, j)$
Intuition:
Aim is to return $h$ such that:

- elements in $A[i : h-1]$ are smaller than pivot
- elements in $A[h+1 : j]$ are at least as large as the pivot
- $A[h]$ contains the pivot element.

P, SSSSSS, LLLLL, C, RRRRRR
(P=pivot, S=smaller, L=larger, C=current, R=remaining)

If C $\geq$ P, then leave it at the same place
If C $<$ P, then swap it with the first L element
At the end swap the pivot with the last S element
Need to be slightly careful in the beginning (when the S part or the L part may be empty).

Need to remember the first L element or the last S element.
(In the algorithm we will remember the last S element).

```
Partition($A, i, j$)
1.  Let $pivot = A[i]$.
2.  Let $h = i$.
3.  For $k = i + 1$ to $j$ {
    (* $A[i]$ is the pivot; $A[i + 1]$,..., $A[h]$ are smaller than pivot.
       $A[h + 1]$,..., $A[k - 1]$ are $\geq$ pivot. *)
        If $A[k] < pivot$, then
            $h = h + 1$
            $swap(A[k], A[h])$
            (that is: $t = A[k]; A[k] = A[h]; A[h] = t;$)
        Endif
    }
4.  $swap(A[h], A[i])$
5.  Return $h$
End
```

At the beginning of the step 3 loop, the following invariant will be true:
$\ldots A[i], A[i+1], \ldots, A[h], A[h+1], \ldots, A[k-1], A[k], \ldots$

- $A[i+1:h]$ are smaller than $A[i]$.
- $A[h+1:k-1]$ are larger than $A[i]$.

Note that initially the invariants are satisfied and each iteration of the for loop maintains the invariant.

5, 7, 2, 1, 9, 7, 4

(Next item: 7; > *pivot*, so leave it as it is.)

5, 7, 2, 1, 9, 7, 4

(Next item: 2; smaller than pivot: swap with 1st larger element)

5, 2, 7, 1, 9, 7, 4

(Next item: 1; smaller than pivot: swap with 1st larger element)

5, 2, 1, 7, 9, 7, 4

(Next item: 9; > *pivot*, so leave it as it is.)

5, 2, 1, 7, 9, 7, 4

(Next item: 7; > *pivot*, so leave it as it is.)

5, 2, 1, 7, 9, 7, 4

(Next item: 4; smaller than pivot: swap with 1st larger element)

5, 2, 1, 4, 9, 7, 7

End of list. Swap the pivot element with last smaller element.

4, 2, 1, 5, 9, 7, 7

```
Quicksort($A, i, j$)
    If $i < j$, then
        $p =$Partition($A, i, j$)
        Quicksort($A, i, p - 1$)
        Quicksort($A, p + 1, j$)
End
```

Correctness:

1. Partition does its job correctly.
2. Assuming Partition does its job correctly, Quicksort does its job correctly.

Complexity:
Time taken is $O(n^2)$
$T(n) \leq T(n-p) + T(p-1) + cn$, for the worst possible $p$ such that $1 \leq p \leq n$.

Guess: $T(r) \leq c_1 r^2$, and prove by induction

$$\begin{aligned}
T(n) &\leq \max_p [c_1(n-p)^2 + c_1(p-1)^2 + cn] \\
&\leq \max_p [c_1[n^2 + p^2 - 2np + p^2 + 1 - 2p] + cn] \\
&\leq \max_p [c_1[n^2 + 1 - 2p(n+1-p)] + cn] \\
&\leq c_1[n^2 - 2n + 1] + cn
\end{aligned}$$

(Here, note that $p(n+1-p)$ is smallest when $p = 1$ or $p = n$)
If one takes $c_1$ such that, $c_1(-2n+1) + cn \leq 0$, then we are done.

Worst Case: Array is already sorted.
$(n-1)+(n-2)+(n-3)+\ldots+1=\Omega(n^2)$.

Best Case: Each round divides the two parts into nearly equal size.
Gives complexity $T(n)=O(n\log n)$.

Advantages: Usually quite quick. No extra array needed.

Randomized Quicksort:
By choosing a random element instead of the first element as the pivot.

```
RandomPartition($A, i, j$)
    $k = rand(i, j)$.
    swap($A[i], A[k]$)
    Partition($A, i, j$)
End

RandomQsort($A, i, j$)
    If $i < j$,
        $p =$RandomPartition($A, i, j$)
        RandomQsort($A, i, p - 1$)
        RandomQsort($A, p + 1, j$)
End
```

$$T(n) \leq \frac{1}{n} \sum_{p=1}^{n} [T(n-p) + T(p-1) + C_1 n]$$

$$T(n) \leq \frac{2}{n} \sum_{p=1}^{n-1} [T(p) + C_1 n]$$

$$nT(n) \leq C_2 n^2 + 2 \sum_{p=1}^{n-1} T(p)$$

Replacing $\leq$ by $=$ in the above line, and using $H$ instead of $T$ one gets,

$$nH(n) = C_2 n^2 + 2 \sum_{p=1}^{n-1} H(p)$$

where $T(n) \leq H(n)$.

$$nH(n) = C_2 n^2 + 2 \sum_{p=1}^{n-1} H(p)$$

Replacing $n$ by $n-1$ in the above equation we get.

$$(n-1)H(n-1) = C_2(n-1)^2 + 2 \sum_{p=1}^{n-2} H(p)$$

Taking the difference of the two equations, we get

$$nH(n) - (n-1)H(n-1) = C_2(2n-1) + 2H(n-1)$$

$$nH(n) = C_2(2n-1) + (n+1)H(n-1)$$

By dividing by $n(n+1)$, we get

$$\begin{aligned}
\frac{H(n)}{n+1} &\le \frac{H(n-1)}{n} + C_3(\frac{1}{n}) \\
&\le \frac{H(n-2)}{n-1} + C_3(\frac{1}{n-1} + \frac{1}{n}) \\
&\le \frac{H(n-3)}{n-2} + C_3(\frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n}) \\
\dots \\
&\le \frac{H(1)}{2} + C_3(\frac{1}{2} + \dots + \frac{1}{n-1} + \frac{1}{n}) \\
&\le C_4 * \log n
\end{aligned}$$

Therefore $T(n) \le H(n) = O(n \log n)$.

## Decision Tree

Consider an algorithm being represented by a tree:
– each node represents the comparison that is made.
– the two child sub-trees represent the continuation of the algorithm based on whether the answer to the comparsion is true of false.

Lower Bound for Sorting Problem

– Assumption: Sorting algorithm works only on the basis of comparisons. That is, it cannot look at the "value" of an item, and only compare different items using $\leq$.
– $\Omega(n \lg n)$ lower bound for comparison based sorting.
– Let $T$ be the decision tree that represents the algorithm for input of size $n$ (that is $n$ elements in the input). In this decision tree, each node represents a comparison, and the left/right sub-tree denotes what the algorithm does based on the result of comparison being true or false.
– Note that the tree must have at least $n!$ leaves, as it must have at least one leaf for each possible order of the inputs.
– Let $h$ denote the height of the comparison tree.
– Thus we have that $2^h \geq n!$.
– Since $\lg(n!) = \Theta(n \lg n)$, we have that $h \in \Omega(n \lg n)$

## Can we do better?

– Note that the lower bound holds, whatever the value inside the arrays may be, as long as one can only do comparisons for the array elements.
– Can we do better if we use something other than comparison among array elements?
– Linear time, indexing ...

## Counting Sort

- Input is an array $A$, with $1 \leq A[i] \leq m$, for all $i$.
- Counting: how many times each element $j$ appears in $A$ via indexing
- Accumulate: count how many elements $\leq j$ appear in the array.
- Sorting: for each $i$, place $A[i]$ in its correct position, and update.

Input: Array $A$ of size $n$, containing only elements $1, 2, \ldots, m$.

```
Counting Sort $(A, m, n)$
1.  For $i = 1$ to $m$ { $Count[i] = 0$; }
2.  For $i = 1$ to $n$ { $Count[A[i]] = Count[A[i]] + 1$; }
    (* Above is counting how many times each element appears *)
3.  For $i = 2$ to $m$ { $Count[i] = Count[i] + Count[i-1]$; }
    (* Above accumulates the count: that is $Count(A[i])$ is the num-
       ber of times elements $A[1], A[2], \ldots, A[i]$ appear. *)
4.  For $i = n$ down to 1 {
            $B[Count[A[i]]] = A[i]$;
            $Count[A[i]] = Count[A[i]] - 1$;
    }
End
```

$B$ is the sorted array.

Example:

| A-index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | 1 | 2 | 1 | 3 | 1 | 3 | 1 | 2 | 2 | 3 |

After step 2:

$Count[1] = 4$

$Count[2] = 3$

$Count[3] = 3$

After step 3:

$Count[1] = 4$

$Count[2] = 7$

$Count[3] = 10$

| A-index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | 1 | 2 | 1 | 3 | 1 | 3 | 1 | 2 | 2 | 3 |

Loop at step 4 – $Count[1] = 4, Count[2] = 7, Count[3] = 10$

– $B[10] = 3$ — $Count[1] = 4, Count[2] = 7, Count[3] = 9$

– $B[7] = 2$ — $Count[1] = 4, Count[2] = 6, Count[3] = 9$

– $B[6] = 2$ — $Count[1] = 4, Count[2] = 5, Count[3] = 9$

– $B[4] = 1$ — $Count[1] = 3, Count[2] = 5, Count[3] = 9$

– $B[9] = 3$ — $Count[1] = 3, Count[2] = 5, Count[3] = 8$

– $B[3] = 1$ — $Count[1] = 2, Count[2] = 5, Count[3] = 8$

– $B[8] = 3$ — $Count[1] = 2, Count[2] = 5, Count[3] = 7$

– $B[2] = 1$ — $Count[1] = 1, Count[2] = 5, Count[3] = 7$

– $B[5] = 2$ — $Count[1] = 1, Count[2] = 4, Count[3] = 7$

– $B[1] = 1$ — $Count[1] = 0, Count[2] = 4, Count[3] = 7$

Correctness:

Complexity: $O(m+n)$ time.
Steps 2 and 4 take time $O(n)$
Steps 1 and 3 take time $O(m)$

Stable: That is, if $A[i] = A[j]$, with $i < j$, and in the sorted array $A[i]$ is stored in $B[i']$ and $A[j]$ is stored in $B[j']$, then $i' < j'$. This is useful when we may want to do some secondary sorting, or maintain original order among equals.

Example: $(2, b), (3, c), (2, d)$
Stable Sort: $(2, b), (2, d), (3, c)$
Non-Stable Sort: $(2, d), (2, b), (3, c)$

## Radix Sort

Counting Sort uses lots of space, if $m >> n$.
To take advantage of a strength of counting sort, one can do sorting bit by bit! The strength we are refering to is being stable.

Sort first on least significant bit, then the next least significant bit, and so on until the most significant bit.
Takes time $O(kn)$, where $k$ is a bound on number of bits in each number.

Radix Sort

```
For $i = 1$ to $k$ {
    Sort (stable) based on $i$-th least significant bit.
}
End
```

Correctness:
Suppose some number $a$ and $b$ are different and the most significant bit on which they differ is the $j$-th least significant bit.
Then, in round $i = j$, they would be put in proper order, and from then on their order will not change! (due to the sorting method used being 'stable').

Complexity:
Takes time $k * n$, if the sorting method used for each bit is counting sort.

## Selection

Input: An array of numbers $A[1], \ldots, A[n]$, and a number $k$.
Output: To find the $k$-th number in sorted order for $A[1], \ldots, A[n]$.

```
Select($A, k, i, j$)
(* Assumption: $i \le k \le j$. *)
    If $i = j$, then return $A[i]$
    If $i < j$, then $p = RandomPartition(A, i, j)$
    If $p = k$, then return $A[p]$
    Else if $k < p$, then return $Select(A, k, i, p - 1)$
    Else if $p < k$, then return $Select(A, k, p + 1, j)$
    Endif
End
```

$$T(n) \leq \frac{1}{n} \sum_{p=1}^{n} [max(T(n-p), T(p-1)) + Cn]$$

Guess: $T(r) \leq 4Cr$.

$$T(n) \leq \frac{1}{n} \sum_{p=1}^{n} [max(4C(n-p), 4C(p-1)) + Cn]$$

$$T(n) \leq Cn + \frac{8C}{n}(\lfloor \frac{n+1}{2} \rfloor + \ldots + n - 1)$$

$$T(n) \leq Cn + \frac{8C}{n}[\frac{n(n-1)}{2} - \frac{\lfloor \frac{n-1}{2} \rfloor * (\lfloor \frac{n-1}{2} \rfloor + 1)}{2}]$$

$$T(n) \leq Cn + \frac{4C}{n}[n(n-1) - (\frac{n}{2})(\frac{n}{2} - 1)]$$

$$T(n) \leq Cn + \frac{4C}{n}[\frac{3n^2}{4}] \leq 4Cn$$

Lower Bound for Selection:
$\Omega(n)$.
Cannot determine the $k$-th element unless we had "looked at" every element.

## Text Search

Input: $T[1],\ldots,T[n]$ and $P[1],\ldots,P[m]$
Output: Yes, if $P[1]\ldots P[m]$ is a substring of $T[1]\ldots T[n]$

Brute Force Approach: Check for every $i$, $1 \leq i \leq n-m$, whether
$T[i] = P[1]$, $T[i+1] = P[2]$, …, $T[i+m-1] = P[m]$

Takes Time $\Theta(mn)$.

Can we do better?
Note that there is a lower bound of $\Omega(n)$ and $\Omega(m)$.

- Rather than comparing $T[i]T[i{+}1]\dots T[i{+}m{-}1]$ with $P[1]P[2]\dots P[m]$, for every $i$, if we can cutdown on the comparisons needed, then we will be able to do faster.
- Suppose we have a hash function *hash* which maps each string of length $m$ to a number. $hash(w)$ is also called fingerprint of $w$.
- Then, we only need to compare $T[i]T[i+1]\dots T[i+m-1]$ with $P[1]P[2]\dots P[m]$, only when the hash function maps the two strings to the same number. If this happens rarely enough, then we can speed up the algorithm!

- We further need to be able to compute hash function incrementally fast that is, we need to calculate $hash(T(i)\ldots T(i{+}m{-}1))$, from $hash(T(i-1)\ldots T(i+m-2))$, $T(i-1)$ and $T(i+m-1))$ fast (rather than doing the computation all over again in each step).
- hash function we use is:

  $hash(b_1 \ldots b_m) = b_1 \ldots b_m \bmod q$, for large enough $q$.

  Then, the probability of matching is $1/q$. So we need to compare once every $q$ times, on expected basis.

Algorithm for binary pattern finding.

```
Search (P[1:m],T[1:n])
1. $m$ = length of $P$; $n$ = length of $T$
2. $q$ = a prime number $> m$.
3. Let $f(1) = 0$; Let $pf = 0$
4. For $j = 1$ to $m$ Do {
        $f(1) = 2 * f(1) + T[j] \bmod q$
        $pf = 2 * pf + P[j] \bmod q$
   }
5. For $i = 1$ to $n - m + 1$ Do {
   5.1. If $f(i) = pf$, then if $T[i : m + i - 1] = P[1 : m]$, then return
        $i$
   5.2. $f(i + 1) = 2 * (f(i) - 2^{m-1} * T[i]) + T[i + m] \bmod q$
   }
End
```

– If hash function is good, then expected number of times we need to do the "comparison", in step 5.1., of the pattern with the text is small.
– There is deterministic $O(m + n)$ algorithm for this problem given by Knuth, Morris and Pratt.