Quick Sort

As the name implies, it is quick, and it is the algorithm generally preferred for sorting.
Basic Ideas

(Another *divide-and-conquer* algorithm)

- Pick an element, say \( P \) (the pivot)
- Re-arrange the elements into 3 sub-blocks,
  1. those less than or equal to \((\leq)\) \( P \) (the **left**-block \( S_1 \))
  2. \( P \) (the only element in the **middle**-block)
  3. those greater than or equal to \((\geq)\) \( P \) (the **right**-block \( S_2 \))
- Repeat the process **recursively** for the **left**- and **right**- sub-blocks. Return \{quicksort(S_1), P, quicksort(S_2)\}. (That is the results of quicksort(S_1), followed by \( P \), followed by the results of quicksort(S_2))
Basic Ideas

Pick a “Pivot” value, P
Create 2 new sets without P

Items smaller than or equal to P

0 31
13 43
26 57

quicksort(S₁)

Items greater than or equal to P

0 13 26 31 43 57

≤ 65 ≤

75 92 81

quicksort(S₂)

≤ ≤

75 81 92

0 13 26 31 43 57

65 75 81 92

Quick Sort
Basic Ideas

S is a set of numbers

$S_1 = \{x \in S - \{P\} | x \leq P\}$

$S_2 = \{x \in S - \{P\} | P \leq x\}$
Basic Ideas

Note:
- The main idea is to find the “right” position for the pivot element $P$.
- After each “pass”, the pivot element, $P$, should be “in place”.
- Eventually, the elements are sorted since each pass puts at least one element (i.e., $P$) into its final position.

Issues:
- How to choose the pivot $P$?
- How to partition the block into sub-blocks?
Implementation

**Algorithm I:**

```c
int partition(int A[], int left, int right);

// sort A[left..right]
void quicksort(int A[], int left, int right)
{
    int q;
    if (right > left)
    {
        q = partition(A, left, right);
        // after 'partition'
        quicksort(A, left, q-1);
        quicksort(A, q+1, right);
    }
}
```

Quick Sort
Implementation

// select A[left] be the pivot element)
int partition(int A[], int left, int right);
{
    P = A[left];
    i = left;
    j = right + 1;
    for(;;) //infinite for-loop, break to exit
    {
        while (A[++i] < P) if (i >= right) break;
        // Now, A[i] ≥ P
        while (A[--j] > P) if (j <= left) break;
        // Now, A[j] ≤ P
        if (i >= j) break; // break the for-loop
        else swap(A[i], A[j]);
    }
    if (j == left) return j;
    swap(A[left], A[j]);
    return j;
}
Example

Input:  65  70  75  80  85  60  55  50  45
P: 65
Pass 1:
(i)  65  70  75  80  85  60  55  50  45
   i       j
   65  45  75  80  85  60  55  70  45
   swap (A[i], A[j])
(ii)  65  45  50  80  85  60  55  75  70
      i       j
      65  45  50  80  85  50  75  70
      swap (A[i], A[j])
(iii)  65  45  50  55  85  60  80  75  70
     i       j
     65  45  50  55  65  80  80  75  70
     swap (A[i], A[j])
(iv)  65  45  50  55  60  85  80  75  70
    i       j
    65  45  50  55  65  80  80  75  70
    swap (A[i], A[j])
(v)  60  45  50  55  65  85  80  75  70
     j       i
     60  45  50  55  65  85  80  75  70
     swap (A[left], A[j])

Items smaller than or equal to 65
Items greater than or equal to 65
Example

Result of Pass 1: 3 sub-blocks:

\[
\begin{array}{cccccccc}
60 & 45 & 50 & 55 & 65 & 85 & 80 & 75 & 70 \\
\end{array}
\]

Pass 2a (left sub-block):

\[
\begin{array}{cccc}
60 & 45 & 50 & 55 \\
\end{array}
\]

\(P = 60\)

\[
\begin{array}{cccc}
i & j \\
60 & 45 & 50 & 55 \\
\end{array}
\]

\(j \leq i \) break

\[
\begin{array}{cccc}
55 & 45 & 50 & 60 \\
\end{array}
\]

swap \((A[\text{left}], A[j])\)

Pass 2b (right sub-block):

\[
\begin{array}{cccc}
85 & 80 & 75 & 70 \\
\end{array}
\]

\(P = 85\)

\[
\begin{array}{cccc}
i & j \\
85 & 80 & 75 & 70 \\
\end{array}
\]

\(i \geq j \) break

\[
\begin{array}{cccc}
70 & 80 & 75 & 85 \\
\end{array}
\]

swap \((A[\text{left}], A[j])\)
Running time analysis

The advantage of this quicksort is that we can sort “in-place”, i.e., *without* the need for a temporary buffer depending on the size of the inputs. (cf. mergesort)

**Partitioning Step:** Time Complexity is $\Theta(n)$.

Recall that quicksort involves *partitioning, and 2 recursive calls.* Thus, giving the basic quicksort relation:

$$T(n) = \Theta(n) + T(i) + T(n-i-1) = cn + T(i) + T(n-i-1)$$

where $i$ is the size of the first sub-block after partitioning.

We shall take $T(0) = T(1) = 1$ as the initial conditions.

To find the solution for this relation, we’ll consider three cases:

1. The Worst-case (?)
2. The Best-case (?)
3. The Average-case (?)

*All depends on the value of the pivot!!*
Running time analysis

**Worst-Case** (Data is sorted already)
- When the pivot is the smallest (or largest) element at partitioning on a block of size \( n \), the result
  - yields one empty sub-block, one element (pivot) in the “correct” place and one sub-block of size \((n-1)\)
  - takes \( \theta(n) \) times.
- Recurrence Equation:
  \[
  \begin{align*}
  T(1) &= 1 \\
  T(n) &= T(n-1) + cn
  \end{align*}
  \]

Solution: \( \theta(n^2) \)

*Worse than Mergesort!!!*
Running time analysis

**Best case:**
- The pivot is in the middle (median) (at each partition step), i.e. after each partitioning, on a block of size $n$, the result
  - yields two sub-blocks of approximately equal size and the pivot element in the “middle” position
  - takes $n$ data comparisons.
- Recurrence Equation becomes
  \[
  \begin{align*}
  T(1) &= 1 \\
  T(n) &= 2T(n/2) + cn
  \end{align*}
  \]
Solution: $\theta(n \log n)$

Comparable to Mergesort!!
Running time analysis

**Average case:**
It turns out the average case running time also is $\Theta(n \log n)$.
We will wait until COMP 271 to discuss the analysis.
So the trick is to select a good pivot

Different ways to select a good pivot.

- First element
- Last element
- Median-of-three elements
  ◆ Pick three elements, and find the median $x$ of these elements. Use that median as the pivot.
- Random element
  ◆ Randomly pick a element as a pivot.
## Different sorting algorithms

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Worst-case time</th>
<th>Average-case time</th>
<th>Space overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Something extra : Selection problem.

Problem statement.

You are given a unsorted array $A[1..n]$ of (distinct) numbers, find an element from the array such that its rank is $i$, i.e., there are exactly $(i-1)$ numbers less than or equal to that element.

Example:

$A=\{5, 1, 2, 3, 12, 20, 30, 6, 14, -1, 0\}$, $i=8$.
Output = 12, since “6, 1, -1, 0, 2, 3, 5” (8-1=7 numbers) are all less than or equal to 12.
Selection problem : a easy answer.

A Easy algorithm.

- Sort A, and return A[i].
- Obviously it works, but it is slow !!! $\Theta(n \log n)$ in average.
- We want something faster.
Selection problem: a ‘faster’ answer.

We can borrow the idea from the partition algorithm.

Suppose we want to find an element of rank \( i \) in \( A[1..n] \).

After the 1\(^{\text{st}}\) partition call (use a random element as pivot):

1. If the return index ‘\( q \)’ = \( i \), then \( A[q] \) is the element we want. (Since there is exactly \( i-1 \) elements smaller than or equal to \( A[q] \)).

2. If the return index ‘\( q \)’ > \( i \), then the target element can NOT be in \( A[q .. \text{right}] \). The target element is rank \( i \) in \( A[1.. q-1] \). \( \rightarrow \) Recursive call with parameters \( (A, 1, q-1, i) \).

3. If the return index ‘\( q \)’ < \( i \), then the target element can NOT be in \( A[1 .. q] \). The target element is rank \( i-q \) in \( A[q+1 .. n] \). \( \rightarrow \) Recursive call with parameters \( (A, q+1,n, i-q) \).
1. $i = q$

2. $i < q$

3. $i > q$

$\Rightarrow A[q] \leq A[i]$ 

Target $= A[q]$

$i-1$ elements $\leq A[q]$ 

Target $A[1..q-1]$, ranked $i$

$i = 1$

Target $A[1..q-1]$, ranked $i$

$\Rightarrow A[q] \leq A[i]$ 

Target $A[q+1..n]$, ranked $i-q$
A ‘faster’ selection algorithm : Codes

\[ A[p..q-1] \leq A[q] \leq A[q+1..r] \]

\[ \text{size of } A[p..q] = k \]

\[
\text{RANDOMIZED-SELECT}(A, p, r, i)
\]

1. if \( p = r \)
2. \quad then return \( A[p] \)
3. \( q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r) \)
4. \( k \leftarrow q - p + 1 \)
5. if \( i = k \) \( \triangleright \) the pivot value is the answer
6. \quad then return \( A[q] \)
7. elseif \( i < k \)
8. \quad then return \( \text{RANDOMIZED-SELECT}(A, p, q - 1, i) \)
9. else return \( \text{RANDOMIZED-SELECT}(A, q + 1, r, i - k) \)

return a element of rank \( i \) in \( A[p..r] \)
Analysis the ‘faster’ answer.

- Though we claim it is a ‘fast’ algorithm, the worst-case running time is $O(n^2)$ (see if you can prove it).
- But the **average-case** running time is only $O(n)$. (Again, we will see the analysis in COMP 271).
- There is an algorithm that runs in $O(n)$ in the worst case, again, we will talk about that in COMP 271.