CSE 12: Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Seventeen
2 Aug 2012

Quicksort.

Quicksort

- The last sorting algorithm we consider is **Quicksort**.
- Quicksort has one of the best performance profiles of all the general-purpose sorting algorithms in the *average case*.
- Like Mergesort, Quicksort is based on the *divide-and-conquer* principle.
- Quicksort differs from Mergesort in how it divides the input array into two pieces.

Quicksort

- The high-level idea of Quicksort is the following:
  - Rearrange ("partition") the input array into a *left part L* and a *right part R* so that:

    everything in the left part $\leq$ everything in the right part.
  - Then, recursively call Quicksort on both the left and right halves.

Quicksort

- Pseudocode:

```
void quicksort (array) {
  If array.length == 1, then do nothing.
  Else:
    Partition array into left part and right part, so that:
      everything in left part ≤ everything in right part.
    quicksort(leftPart);
    quicksort(rightPart);
}
```

Partitioning

- The key to Quicksort is the **`partition`** function, which needs to operate in $O(n)$ time.
- **`partition(array)`** works by picking a *pivot* element x from **`array`**.
  - Left part contains elements $\leq x$.
  - Right part contains elements $\geq x$.
- The simplest implementations choose the *first* element of **`array`** as the pivot.
- Better-performing implementations choose a *random* element of **`array`**.

Partitioning

- The `partition` method will work as follows:

```
void partition (array) {
  pivot = pickPivot(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

- This procedure will effectively move all elements ≤ `pivot` to the *left*, and all elements ≥ `pivot` to the *right*.

Partitioning

- Let’s try an example where we select the pivot to just be the array’s *first element*:

6 1 4 3 8 7 2 5

☝ ☝

i = -1 **pivot = 6** **j = 8**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

6 1 4 3 8 7 2 5

☝ ☝

i = 0 **pivot = 6** **j = 8**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

6 1 4 3 8 7 2 5

☝ ☝

i = 0 **pivot = 6** **j = 7**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 8 7 2 6

i = 0 pivot = 6 j = 7

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

```
5 1 4 3 8 7 2 6
  ☝           ☝

i = 1      pivot = 6      j = 7
```

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 8 7 2 6

☝ ☝

i = 2　　**pivot = 6**　　**j = 7**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 8 7 2 6

i = 3 **pivot = 6** **j = 7**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 8 7 2 6

i = 4 **pivot = 6** **j = 7**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 8 7 2 6

☝ ☝

i = 4 **pivot = 6** **j = 6**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let’s try an example where we select the pivot to just be the array’s *first element*:

5 1 4 3 2 7 8 6

☝ ☝

i = 4 pivot = 6 j = 6

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 2 7 8 6

☝ ☝

i = 5 **pivot = 6** **j = 6**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 2 7 8 6

i = 5 pivot = 6 j = 5

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 2 7 8 6

☝ ☝

i = 5 **pivot = 6** **j = 4**

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 2 7 8 6

☝ ☝

i = 5 pivot = 6 j = 4

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

5 1 4 3 2 7 8 6

☝ ☝

i = 5 pivot = 6 j = 4

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

Partitioning

- Let's try an example where we select the pivot to just be the array's *first element*:

Left part   Right part

5 1 4 3 2 7 8 6

☝ ☝

`i = 5`   `pivot = 6`   `j = 4`

Done.

```
void partition (array) {
  pivot = pickRandomElement(array);
  Set i = -1
  Set j = N
  while i < j:
    Increment i until array[i] ≥ pivot.
    Decrement j until array[j] ≤ pivot.
    If i < j, then swap array[i] and array[j].
}
```

The partition method also has the nice side effect that the final value of `j` tells us the *right-most* element in the *left part.*

Quicksort

- Example:

`6 1 4 3 8 7 2 5`

Partition.

Quicksort

- Example:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6
```

Recurse.

Left part Right part

Quicksort

- Example:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6

5 1 4 3 2          7 8 6
```

Partition.

Quicksort

- Example:

`6 1 4 3 8 7 2 5`

`5 1 4 3 2 7 8 6`

`5 1 4 3 2` `7 8 6`

`2 1 4 3 5` `6 8 7` Recurse.

Left part Right part Left part Right part

Quicksort

- Example:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6

5 1 4 3 2          7 8 6
2 1 4 3 5          6 8 7

2 1 4 3    5     6     8 7
```

Partition.

Quicksort

- Example:

6 1 4 3 8 7 2 5

5 1 4 3 2 7 8 6

5 1 4 3 2   7 8 6

2 1 4 3 5   6 8 7

2 1 4 3   5   6   8 7

1 2 4 3   5   6   7 8   Recurse.

Quicksort

- Example:

```
      6 1 4 3 8 7 2 5
      5 1 4 3 2 7 8 6

5 1 4 3 2         7 8 6
2 1 4 3 5         6 8 7

2 1 4 3    5    6    8 7
1 2 4 3    5    6    7 8

1   2 4 3  5    6  7   8
```

Partition.

Quicksort

- Example:

```
        6 1 4 3 8 7 2 5
        5 1 4 3 2 7 8 6

   5 1 4 3 2          7 8 6
   2 1 4 3 5          6 8 7

  2 1 4 3     5     6     8 7
  1 2 4 3     5     6     7 8

1    2 4 3    5     6   7    8
1    2 4 3    5     6   7    8
```

Recurse.

Quicksort

- Example:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6

5 1 4 3 2          7 8 6
2 1 4 3 5          6 8 7

2 1 4 3    5     6     8 7
1 2 4 3    5     6     7 8

1    2 4 3    5     6   7    8
1    2 4 3    5     6   7    8

1    2    4 3    5     6   7    8
```

Partition.

Quicksort

- Example:

```
            6 1 4 3 8 7 2 5
            5 1 4 3 2 7 8 6

   5 1 4 3 2            7 8 6
   2 1 4 3 5            6 8 7

 2 1 4 3    5      6       8 7
 1 2 4 3    5      6       7 8

1    2 4 3  5      6    7    8
1    2 4 3  5      6    7    8

1   2   4 3 5      6    7    8
1   2   3 4 5      6    7    8
```

Recurse.

Quicksort

- Example:

```
            6 1 4 3 8 7 2 5
            5 1 4 3 2 7 8 6

   5 1 4 3 2          7 8 6
   2 1 4 3 5          6 8 7

 2 1 4 3    5      6     8 7
 1 2 4 3    5      6     7 8

1   2 4 3   5      6   7   8
1   2 4 3   5      6   7   8

1  2   4 3  5      6   7   8
1  2   3 4  5      6   7   8
1  2  3  4  5      6   7   8
```

Done.

Quicksort

- We can also do this *in-place*:

```
6 1 4 3 8 7 2 5
```

Done.

Quicksort

- We can also do this *in-place*:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6
```

Done.

Quicksort

- We can also do this *in-place*:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6
2 1 4 3 5 6 8 7
```

Done.

Quicksort

- We can also do this *in-place*:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6
2 1 4 3 5 6 8 7
1 2 4 3 5 6 7 8
```

5 and 6 are both 1-element lists -- stop recursion.

Done.

Quicksort

- We can also do this *in-place*:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6
2 1 4 3 5 6 8 7
1 2 4 3 5 6 7 8
1 2 4 3 5 6 7 8
```

Done.

Quicksort

- We can also do this *in-place*:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6
2 1 4 3 5 6 8 7
1 2 4 3 5 6 7 8
1 2 4 3 5 6 7 8
1 2 3 4 5 6 7 8
```

Done.

Quicksort

- We can also do this *in-place*:

```
6 1 4 3 8 7 2 5
5 1 4 3 2 7 8 6
2 1 4 3 5 6 8 7
1 2 4 3 5 6 7 8
1 2 4 3 5 6 7 8
1 2 3 4 5 6 7 8
```

Done.

Quicksort

- The version of Quicksort just demonstrated operates *in-place*, but it is not *stable*.
  - Alternative implementations are *stable*, but do not operate in-place.

Quicksort

- With Quicksort, *all* the sorting all happens "on the way down" the stack of recursive calls.
  - As soon as every call to Quicksort has reached the base case, the array is *sorted*.
- Contrast this with Mergesort, in which the *merging* takes place "on the way back up" the stack of recursive calls.
  - As soon as every call to Mergesort has reached the base case, not even a single element has been re-arranged.

Mergesort

- Example: First stage: recursively divide until we reach the base case.

6 1 4 3 8 7 2 5

Split list and recurse.

```
void mergesort (array) {
  If array.length == 1, then do nothing.
  Else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```

Mergesort

- Example: First stage: recursively divide until we reach the base case.

6 1 4 3 8 7 2 5

6 1 4 3 **8 7 2 5**

Split list and recurse.

```
void mergesort (array) {
  If array.length == 1, then do nothing.
  Else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```

Mergesort

- Example: First stage: recursively divide until we reach the base case.

```
            6 1 4 3 8 7 2 5                  Split list and recurse.

      6 1 4 3                8 7 2 5         Split list and recurse.

   6 1         4 3         8 7         2 5   Split list and recurse.

 6     1     4     3     8     7     2     5
```

```
void mergesort (array) {
  If array.length == 1, then do nothing.
  Else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```

Mergesort

- Example:

Each of these is a “list” (size 1) passed to a recursive call to Mergesort.

6 1 4 3 8 7 2 5

```
void mergesort (array) {
  If array.length == 1, then do nothing.
  Else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```

Mergesort

- Example:

Second stage: merge each pair of sorted sub-lists.

1 6 3 4 7 8 2 5

Merge the two sub-lists.

6 1 4 3 8 7 2 5

```
void mergesort (array) {
  If array.length == 1, then do nothing.
  Else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```

Mergesort

- Example: Second stage: merge each pair of sorted sub-lists.

1 3 4 6   2 5 7 8   Merge the two sub-lists.

1 6   3 4   7 8   2 5

6   1   4   3   8   7   2   5

```
void mergesort (array) {
  If array.length == 1, then do nothing.
  Else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```

Mergesort

- Example:

Second stage: merge each pair of sorted sub-lists.

1 2 3 4 5 6 7 8

Merge the two sub-lists.

1 3 4 6   2 5 7 8

1 6   3 4   7 8   2 5

6   1   4   3   8   7   2   5

```
void mergesort (array) {
  If array.length == 1, then do nothing.
  Else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```

Mergesort

- Example:

Done.

```
              1 2 3 4 5 6 7 8

      1 3 4 6           2 5 7 8

   1 6     3 4      7 8      2 5

  6   1   4   3   8   7   2   5
```

```
void mergesort (array) {
  If array.length == 1, then do nothing.
  Else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```

Quicksort

- The time cost of Quicksort in the *average case* differs substantially from the *worst case*.
- In the average case, the `partition` procedure splits the array into equal-sized parts.
  - This results in a recursion depth of $O(\log n)$.
- At each level of recursion, the entire array must be "touched" (during partitioning), so *n*.
- In total, Quicksort is $n * O(\log n) = O(n \log n)$.

Quicksort

- In the *worst case*, the `partition` procedure splits the array into a 1-element part, and a n-1-element part.
- If this occurs throughout the entire recursion stack, then the recursion depth will be $O(n)$ instead of $O(\log n)$.
- Since every element must still be "touched" at each level of recursion, this results in $O(n) * O(n) = O(n^2)$ operations.
  - Hence, in the worst case, Quicksort is no better than insertion/selection sort.

Quicksort

- This worst case is realized if (a) the input array is already sorted and (b) we always choose the first element to be the pivot.
- Example:

Left part Right part

```
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
```

n levels deep

Quicksort

- To prevent this worst-case performance from happening, practical implementations of Quicksort pick the pivot element *randomly*.
  - This ensures that, on a list that is already sorted, Quicksort still gives $O(n \log n)$ performance.

This is the last slide of the course.