Quick Sort

David Casperson

[1]Department of Computer Science
University of Northern BC

2018-09-28 / CPSC 200 Lecture

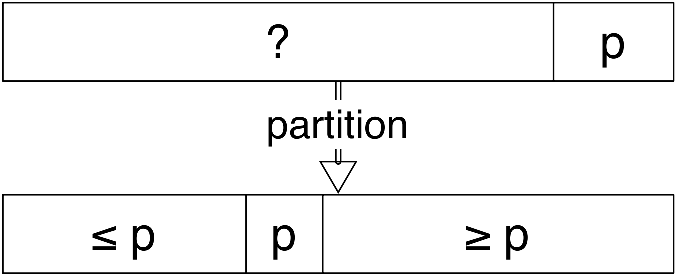
David Casperson

Outline

- We know that we can sort in $\mathrm{O}(n^2)$ time.
- We know from sub-sequence sum algorithms that if we can split the problem into two half-size problems we win

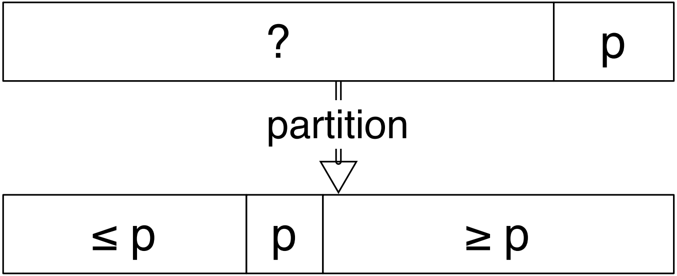
- We know that we can sort in $\mathrm{O}(n^2)$ time.
- We know from sub-sequence sum algorithms that if we can split the problem into two half-size problems we win because $2(n/2)^2 < n^2$.
- *Partitioning* splits the problem approximately into half.

- The Goal of the Partition algorithm



- The problem with the Partition algorithm . . .

- The Goal of the Partition algorithm



- The problem with the Partition algorithm . . . we don't know how to choose p.
- Good choices of p lead to $\Theta(n \log n)$
- Bad choices lead to $\Theta(n^2)$.

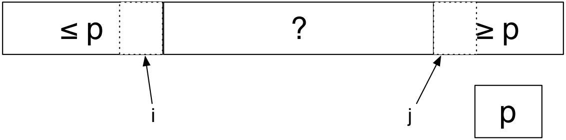
The Partition Algorithm
Selecting the pivot

- pivot selection is very important
- *Weiss* uses best of three
- the *Weiss* method puts pivots in place
- the *Weiss* method works well with nearly sorted data

The Partition Algorithm

The picture

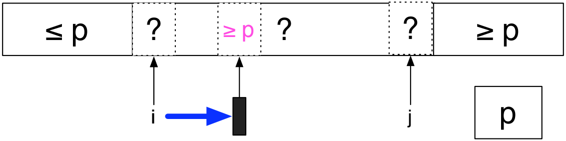
the outer loop




```
while (i<j) {
 while (c.compare(data[++i],p)<0);
 while (c.compare(p,data[--j])<0);
 if (i<j) swap(data, i, j);
}
```

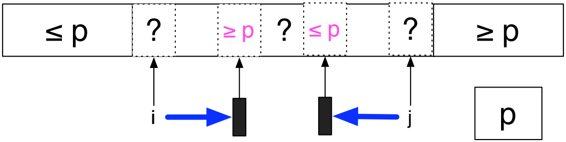
The Partition Algorithm
The picture

moving i



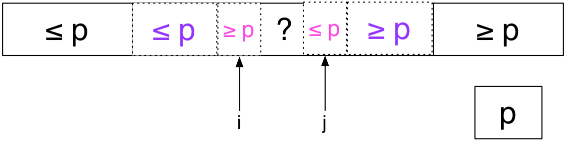

```
while (i<j) {
 while (c.compare(data[++i],p)<0);
 while (c.compare(p,data[--j])<0);
 if (i<j) swap(data, i, j);
}
```

moving j




```
while (i<j) {
 while (c.compare(data[++i],p)<0);
 while (c.compare(p,data[--j])<0);
 if (i<j) swap(data, i, j);
}
```

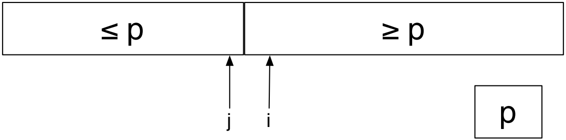
swapping the out of place pair




```
while (i<j) {
 while (c.compare(data[++i],p)<0);
 while (c.compare(p,data[--j])<0);
 if (i<j) swap(data, i, j);
}
```

The Partition Algorithm
The picture

afterward. . .




```
while (i<j) {
 while (c.compare(data[++i],p)<0);
 while (c.compare(p,data[--j])<0);
 if (i<j) swap(data, i, j);
}
```

.

The Partition Algorithm
The details

- elements equal to the pivot are critical
  swap them to make sure split is equal
- sentinels are critical
  let pivot selection put them in place
- constants are very good
- exploits modern cache well.

The overall strategy
code

```
public static <E>
void qsort_rec(E [] data, int b, int e,
                Comparator<E> c)   {
    if (e-b < cutoff) return ;
    int m = (b+e)/2 ;
    sort3(data, b, m, e-1,  c) ;
    swap(data, m, e-1) ;
    m = partition(data,b,e-1,data[e-1], c) ;
    swap(data, m, e-1) ;
    qsort_rec(data,b,m,c) ;
    qsort_rec(data,m+1,e,c) ;
    return ;
}
```

Need a base case!

The overall strategy

code


```
public static <E>
void qsort_rec(E [] data, int b, int e,
                  Comparator<E> c)   {
    if (e-b < cutoff) return ;
    int m = (b+e)/2 ;
    sort3(data, b, m, e-1,  c) ;
    swap(data, m, e-1) ;
    m = partition(data,b,e-1,data[e-1], c) ;
    swap(data, m, e-1) ;
    qsort_rec(data,b,m,c) ;
    qsort_rec(data,m+1,e,c) ;
    return ;
}
```

Pick a pivot and move it out of the way.

The overall strategy

code

```
public static <E>
void qsort_rec(E [] data, int b, int e,
                    Comparator<E> c)   {
    if (e-b < cutoff) return ;
    int m = (b+e)/2 ;
    sort3(data, b, m, e-1,  c) ;
    swap(data, m, e-1) ;
    m = partition(data,b,e-1,data[e-1], c)  ;
    swap(data, m, e-1) ;
    qsort_rec(data,b,m,c) ;
    qsort_rec(data,m+1,e,c) ;
    return ;
}
```

Partition and move the pivot back.

```
public static <E>
void qsort_rec(E [] data, int b, int e,
                    Comparator<E> c)   {
    if (e-b < cutoff) return ;
    int m = (b+e)/2 ;
    sort3(data, b, m, e-1,  c) ;
    swap(data, m, e-1) ;
    m = partition(data,b,e-1,data[e-1], c) ;
    swap(data, m, e-1) ;
    qsort_rec(data,b,m,c) ;
    qsort_rec(data,m+1,e,c) ;
    return ;
}
```

Solve one subproblem.

The overall strategy
code


```
public static <E>
void qsort_rec(E [] data, int b, int e,
                  Comparator<E> c)   {
    if (e-b < cutoff) return ;
    int m = (b+e)/2 ;
    sort3(data, b, m, e-1,  c) ;
    swap(data, m, e-1) ;
    m = partition(data,b,e-1,data[e-1], c) ;
    swap(data, m, e-1) ;
    qsort_rec(data,b,m,c) ;
    qsort_rec(data,m+1,e,c) ;
    return ;
}
```

Solve the other subproblem.

The overall strategy
code


```
public static <E>
void qsort_rec(E [] data, int b, int e,
                Comparator<E> c)   {
    if (e-b < cutoff) return ;
    int m = (b+e)/2 ;
    sort3(data, b, m, e-1,  c) ;
    swap(data, m, e-1) ;
    m = partition(data,b,e-1,data[e-1], c) ;
    swap(data, m, e-1) ;
    qsort_rec(data,b,m,c) ;
    qsort_rec(data,m+1,e,c) ;
    return ;
}
```

Done.

The Overall Strategy
managing space

To manage space

- solve the smaller problem first
- solve the larger problem non-recursively
- This leads to a stack depth of at most $\lceil \log_2 n \rceil$

The Overall Strategy
managing space


```
public static <E>
void qsort_rec(E [] data, int b, int e,
                    Comparator<E> c)
{
     if (e-b < cutoff) return;

     int m = (b+e)/2 ;
     sort3(data, b, m, e-1,  c) ;
     swap(data, m, e-1) ;
     m = partition(data,b,e-1,data[e-1], c) ;
     swap(data, m, e-1) ;
     if(2*m>b+e) { qsort_rec(data, m+1,e, c) ; e=m  ;}
     else        { qsort_rec(data, b,  m, c) ; b=m+1;}

}
```

The Overall Strategy

managing space

```
public static <E>
void qsort_rec(E [] data, int b, int e,
                    Comparator<E> c)
{

    while (!(e-b < cutoff)) {
    int m = (b+e)/2 ;
    sort3(data, b, m, e-1,  c) ;
    swap(data, m, e-1) ;
    m = partition(data,b,e-1,data[e-1], c) ;
    swap(data, m, e-1) ;
    if(2*m>b+e) { qsort_rec(data, m+1,e, c) ; e=m  ;}
    else        { qsort_rec(data, b,  m, c) ; b=m+1;}
    }
    return ;
}
```

The Overall Strategy

managing space

```
public static <E>
void qsort_rec(E [] data, int b, int e,
                    Comparator<E> c)
{

    while (!(e-b < cutoff)) {
    int m = (b+e)/2 ;
    sort3(data, b, m, e-1,  c) ;
    swap(data, m, e-1) ;
    m = partition(data,b,e-1,data[e-1], c) ;
    swap(data, m, e-1) ;
    if(2*m>b+e) { qsort_rec(data, m+1,e, c) ; e=m  ;}
    else        { qsort_rec(data, b,  m, c) ; b=m+1;}
    }
    return ;
}
```

The Overall Strategy

managing space

```
public static <E>
void qsort_rec(E [] data, int b, int e,
                    Comparator<E> c)
{

     while (!(e-b < cutoff)) {
     int m = (b+e)/2 ;
     sort3(data, b, m, e-1,  c) ;
     swap(data, m, e-1) ;
     m = partition(data,b,e-1,data[e-1], c) ;
     swap(data, m, e-1) ;
     if(2*m>b+e) { qsort_rec(data, m+1,e, c) ; e=m  ;}
     else        { qsort_rec(data, b,  m, c) ; b=m+1;}
     }
     return ;
}
```

The Overall Strategy

Avoiding n^2 time

To manage time

The Overall Strategy
Avoiding n^2 time

To manage time

- abandon ship when we clearly aren't making progress.
- if we are more than about 40 problems deep, something has gone wrong.

Quick sort:

- uses partitioning
- uses sentinels
- works well with hardware
- requires care

Quick Sort Summary

Report Card

- $T_{worst}(n) = \Theta(n^2)$

- $T_{worst}(n) = \Theta(n^2)$
  happens when continual bad partitions happen

- $T_{worst}(n) = \Theta(n^2)$
  happens when continual bad partitions happen
- $T_{ave}(n) = \Theta(n \log n)$
  almost all the time

Quick Sort Summary
Report Card

- $T_{worst}(n) = \Theta(n^2)$
  happens when continual bad partitions happen
- $T_{ave}(n) = \Theta(n \log n)$
  almost all the time
- $\mathrm{O}(\log n)$ extra storage for recursion,

- $T_{worst}(n) = \Theta(n^2)$
  happens when continual bad partitions happen
- $T_{ave}(n) = \Theta(n \log n)$
  almost all the time
- $O(\log n)$ extra storage for recursion, with care.

Quick Sort Summary
Report Card

- $T_{worst}(n) = \Theta(n^2)$
  happens when continual bad partitions happen
- $T_{ave}(n) = \Theta(n \log n)$
  almost all the time
- $\mathrm{O}(\log n)$ extra storage for recursion, with care.

Quick Sort Summary

Report Card

- $T_{worst}(n) = \Theta(n^2)$
  happens when continual bad partitions happen
- $T_{ave}(n) = \Theta(n \log n)$
  almost all the time
- $\mathrm{O}(\log n)$ extra storage for recursion, with care.
- not stable.
- very fast

The End