FoCS Lecture 5: Sorting
Anil Madhavapeddy & Amanda Prorok
21st Oct 2019
Applications of sorting

- fast search
- fast merging
- finding duplicates
- inverting tables
- graphics algorithms
Applications of sorting

• fast search
• fast merging
• finding duplicates
• inverting tables
• graphics algorithms

Once a set of items is sorted, it simplifies many other problems in computer science.
Complexity of Comparison Sort?

- typically count the number of comparisons $C(n)$
- there are $n!$ permutations of $n$ elements
- each comparison eliminates *half* of the permutations $2^{C(n)} \geq n!$
- therefore $C(n) \geq \log(n!) \approx n \log n - 1.44n$
- The lower bound of comparison is $O(n \log n)$
Common sorting algorithms

We begin by examining three in detail:

- Insertion sort
- Quicksort
- Mergesort
Insertion Sort
Insertion Sort

```ocaml
# let rec ins = function
  | x, [] -> [x]
  | x, y::ys ->
    if x <= y then
      x :: y :: ys
    else
      y :: ins (x, ys)

# let rec insort = function
  | [] -> []
  | x::xs -> ins (x, insort xs)
```
Insertion Sort

# let rec ins = function
  | x, [] -> [x]
  | x, y::ys ->
    if x <= y then
      x :: y :: ys
    else
      y :: ins (x, ys)

# let rec insort = function
  | [] -> []
  | x::xs -> ins (x, insort xs)

Input is inserted in the output in the right place to be sorted
Insertion Sort

# let rec ins = function
| x, [] -> [x]
| x, y::ys ->
  if x <= y then
    x :: y :: ys
  else
    y :: ins (x, ys)

# let rec insort = function
| [] -> []
| x::xs -> ins (x, insort xs)

Input is inserted in the output in the right place to be sorted
Then continue to process the remainder of the input
Insertion Sort

- Items from input are copied to the output
- Inserted in order, so the output is always sorted
Insertion Sort

- Items from input are copied to the output
- Inserted in order, so the output is always sorted

Complexity is $O(n^2)$ comparisons vs the theoretical best case of $O(n \log n)$
Quicksort
Quicksort

- Choose a *pivot element* \( a \)
- **Divide:** partition the input into two sublists
  - those at most \( a \) in value
  - those *exceeding* \( a \)
- **Conquer:** using recursive calls to sort sublists
- **Combine:** sorted lists by appending them
Quicksort

```ocaml
# let rec quick = function
  | [] -> []
  | [x] -> [x]
  | a::bs ->
    let rec part = function
      | (l, r, []) -> (quick l) @ (a :: quick r)
      | (l, r, x::xs) ->
        if (x <= a) then
          part (x::l, r, xs)
        else
          part (l, x::r, xs)
    in
    part ([], [], bs)
```
Quicksort

```ocaml
# let rec quick = function
  | [] -> []
  | [x] -> [x]
  | a::bs ->
    let rec part = function
      | (l, r, []) -> (quick l) @ (a :: quick r)
      | (l, r, x::xs) ->
        if (x <= a) then
          part (x::l, r, xs)
        else
          part (l, x::r, xs)
    in
    part ([], [], bs)
```

“Divide”
# let rec quick = function
| [] -> []
| [x] -> [x]
| a::bs ->
  let rec part = function
  | (l, r, []) -> (quick l) @ (a :: quick r)
  | (l, r, x::xs) ->
    if (x <= a) then
      part (x::l, r, xs)
    else
      part (l, x::r, xs)
  in
  part ([], [], bs)
Quick sort

```
# let rec quick = function
| [] -> []
| [x] -> [x]
| a::bs ->
  let rec part = function
    | (l, r, []) -> (quick l) @ (a :: quick r)
    | (l, r, x::xs) ->
      if (x <= a) then
        part (x::l, r, xs)
      else
        part (l, x::r, xs)
  in
  part ([], [], bs)
```

“Divide”

“Conquer”

“Combine”
Quicksort

# let rec quick = function
  | []   -> []
  | [x]  -> [x]
  | a::bs ->
    let rec part = function
      | (l, r, [])   -> (quick l) @ (a :: quick r)
      | (l, r, x::xs) ->
        if (x <= a) then
          part (x::l, r, xs)
        else
          part (l, x::r, xs)
    in
    part ([], [], bs)

Complexity is $O(n \log n)$ in the average case
Quicksort

Complexity is $O(n \log n)$ in the average case but $O(n^2)$ in the worst case!
# let rec quik = function
|  ( [],  sorted) ->  sorted
|  ([x], sorted) -> x::sorted
|  a::bs,  sorted ->
  let rec part = function
  |  l,  r,  [] ->  quik (l,  a ::  quik (r,  sorted))
  |  l,  r,  x::xs ->
    if  x <=  a then
      part (x::l,  r,  xs)
    else
      part (l,  x::r,  xs)
  in
  part ( [],  [],  bs)
Comparing both quicksorts

let rec quick = function
| [] -> []
| [x] -> [x]
| a::bs ->
  let rec part = function
  | (l, r, []) ->
    (quick l) @ (a :: quick r)
  | (l, r, x::xs) ->
    if (x <= a) then
      part (x::l, r, xs)
    else
      part (l, x::r, xs)
  in
  part ([], [], bs)

let rec quik = function
| [], sorted -> sorted
| [x], sorted -> x::sorted
| a::bs, sorted ->
  let rec part = function
  | (l, r, []) ->
    quik (l, a :: quik (r, sorted))
  | (l, r, x::xs) ->
    if x <= a then
      part (x::l, r, xs)
    else
      part (l, x::r, xs)
  in
  part ([], [], bs)
Comparing both quicksorts

let rec quik = function
  | [], sorted -> sorted
  | [x], sorted -> x::sorted
  | a::bs, sorted ->
    let rec part = function
      | l, r, [] ->
        quik (l, a :: quik (r, sorted))
      | l, r, x::xs ->
        if x <= a then
          part (x::l, r, xs)
        else
          part (l, x::r, xs)
    in
    part ([], [], bs)

let rec quick = function
  | [] -> []
  | [x] -> [x]
  | a::bs ->
    let rec part = function
      | (l, r, []) ->
        (quick l) @ (a :: quick r)
      | (l, r, x::xs) ->
        if (x <= a) then
          part (x::l, r, xs)
        else
          part (l, x::r, xs)
    in
    part ([], [], bs)

Call “quick” twice and then append results

Call “quik” once, cons “a” to it, then call “quik” again
Mergesort
Merge Two Lists

# let rec merge = function
  | [], ys -> ys
  | xs, [] -> xs
  | x::xs, y::ys ->
    if x <= y then
      x :: merge (xs, y::ys)
    else
      y :: merge (x::xs, ys)
Merge Two Lists

• Does at most \((m + n - 1)\) comparisons where \(m\) and \(n\) are length of input lists

• Fast if lists are roughly equal and >1 length

Useful as the basis for several other divide-and-conquer algorithms.

```ocaml
# let rec merge = function
    | [], ys -> ys
    | xs, [] -> xs
    | x::xs, y::ys ->
      if x <= y then
        x :: merge (xs, y::ys)
      else
        y :: merge (x::xs, ys)
```

Top down mergesort

```haskell
# let rec tmergesort = function
  | [] -> []
  | [x] -> [x]
  | xs ->
    let k = List.length xs / 2 in
    let l = tmergesort (take (xs, k)) in
    let r = tmergesort (drop (xs, k)) in
    merge (l, r)
```
Top down mergesort

```ocaml
# let rec tmergesort = function
  | [] -> []
  | [x] -> [x]
  | xs ->
    let k = List.length xs / 2 in
    let l = tmergesort (take (xs, k)) in
    let r = tmergesort (drop (xs, k)) in
    merge (l, r)
```

- Unlike quicksort, no need to pick a pivot
- Count half the list and divide using `take` and `drop`
Top down mergesort

```ocaml
# let rec tmergesort = function
| []        -> []
| [x]       -> [x]
| xs        ->
  let k = List.length xs / 2 in
  let l = tmergesort (take (xs, k)) in
  let r = tmergesort (drop (xs, k)) in
  merge (l, r)
```

- Unlike quicksort, no need to pick a pivot
- Count half the list and divide using `take` and `drop`
Top down mergesort

```ocaml
# let rec tmergesort = function
    | [] -> []
    | [x] -> [x]
    | xs ->
        let k = List.length xs / 2 in
        let l = tmergesort (take (xs, k)) in
        let r = tmergesort (drop (xs, k)) in
        merge (l, r)
```

- Complexity of mergesort is $O(n \log n)$
- But unlike quicksort, is always that even in the worst case.
- So why not always use mergesort?
Sorting through sorting algorithms

Optimal is $O(n \log n)$ comparisons
Sorting through sorting algorithms

Optimal is $O(n \log n)$ comparisons

**Insertion sort**: simple to code, quadratic complexity
**Quicksort**: fast on average, quadratic complexity in worst case
**Mergesort**: optimal in theory, often slower than quicksort in practice
Sorting through sorting algorithms

Optimal is $O(n \log n)$ comparisons

**Insertion sort:** simple to code, quadratic complexity
**Quicksort:** fast on average, quadratic complexity in worst case
**Mergesort:** optimal in theory, often slower than quicksort in practise

Match the algorithm to the application
Exercises

Optimal is $O(n \log n)$ comparisons

**Insertion sort:** simple to code, quadratic complexity
**Quicksort:** fast on average, quadratic complexity in worst case
**Mergesort:** optimal in theory, often slower than quicksort in practise

**Work through selection sort and bubblesort,** and examine the complexity and runtime tradeoffs of their approaches