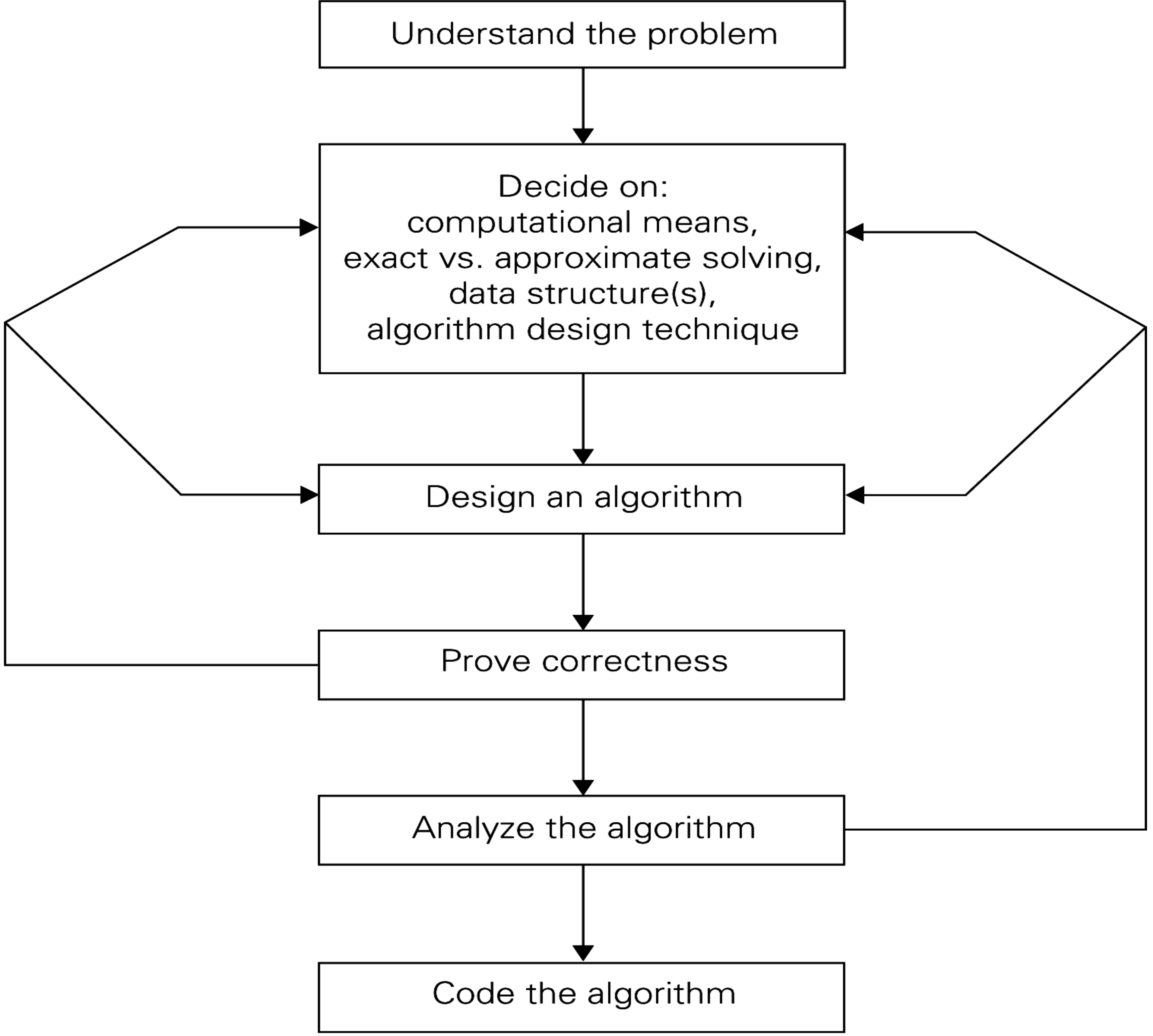
JICSCI803
Algorithms and Data Structures
2020

Highlights of Lecture 04

Sort algorithms

Algorithm Design and Analysis Process



Algorithm Analysis Framework

Measuring an input's size
Measuring running time
Orders of growth (of the algorithm's efficiency function)
Worst-base, best-case and average-case efficiency

Sort Algorithms

Problem:

Re-arrange an array A of n numbers to be in non-descending order.

Example

Before sorting:

$$A = \{50, 30, 40, 80, 70, 10, 90, 60\}.$$

After:

$$A = \{10, 30, 40, 50, 60, 70, 80, 90\}.$$

Selection Sort:

- Scanning the entire given list to find its smallest element and exchange it with the first element; repeat.

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

- Selection Sort, an example.

Bubble Sort Algorithm

The algorithm performs $n - 1$ *passes*. After the i-th pass ($i \leq n - 1$), it is guaranteed that the i-th largest number is moved to its correct position.

Example

Originally: $A = \{50, 30, 40, 80, 70, 10, 90, 60\}$.

Pass 1

{30, 50, 40, 80, 70, 10, 90, 60}
{30, 40, 50, 80, 70, 10, 90, 60}
{30, 40, 50, 80, 70, 10, 90, 60}
{30, 40, 50, 70, 80, 10, 90, 60}
{30, 40, 50, 70, 10, 80, 90, 60}
{30, 40, 50, 70, 10, 80, 90, 60}
{30, 40, 50, 70, 10, 80, 60, 90}

The largest number 90 now appears at the end.

Bubble Sort Algorithm

Example (cont.)

From the previous pass: $A = \{30, 40, 50, 70, 10, 80, 60, 90\}$.

Pass 2

```
{30, 40, 50, 70, 10, 80, 60,   90}
{30, 40, 50, 70, 10, 80, 60,   90}
{30, 40, 50, 70, 10, 80, 60,   90}
{30, 40, 50, 10, 70, 80, 60,   90}
{30, 40, 50, 10, 70, 80, 60,   90}
{30, 40, 50, 10, 70, 60, 80,   90}
```

The second largest number 80 is now at the right place.

Bubble Sort Algorithm

algorithm BubbleSort($A[1..n]$)

```
1. for $i = 1$ to $n - 1$   // the $i$-th pass
2.     for $j = 1$ to $n - i$
3.         if $A[j] > A[j + 1]$
4.             swap $A[j]$ with $A[j + 1]$
```

Analysis of bubble sort

- The i-th ($i \leq n-1$) pass performs $n-i$ comparisons and at most $n-i$ swaps. Hence, the pass takes $O(n-i)$ time.
- Let $f(n)$ be the total cost of bubble sort. Then:

$$\begin{aligned} f(n) &= \sum_{i=1}^{n-1} O(n-i) = \sum_{i=1}^{n-1} O(i) \\ &= O\left(\sum_{i=1}^{n-1} i\right) \\ &= O(n(n-1)/2) = O(n^2). \end{aligned}$$

Merge Sort Algorithm

We will improve the running time dramatically to $O(n \log n)$ by an algorithm called *merge sort*. This algorithm is motivated by the observation that two sorted arrays can be merged in linear time.

Example

Given

$$A_1 = \{30, 40, 50, 80\}, \quad A_2 = \{10, 60, 70, 90\}$$

we want to merge them into

$$A_{merge} = \{10, 30, 40, 50, 60, 70, 80, 90\}$$

in $O(|A_1| + |A_2|)$ time.

Merge Two Sorted Arrays

To merge A_1 and A_2, we scan their elements *synchronously* in ascending order, and move the elements scanned to A_{merge}.

Example (cont.)

1. $A_1 = \{30, 40, 50, 80\}, A_2 = \{10, 60, 70, 90\}, A_{merge} = \{\}$
2. $A_1 = \{30, 40, 50, 80\}, A_2 = \{60, 70, 90\}, A_{merge} = \{10\}$
3. $A_1 = \{40, 50, 80\}, A_2 = \{60, 70, 90\}, A_{merge} = \{10, 30\}$
4. $A_1 = \{50, 80\}, A_2 = \{60, 70, 90\}, A_{merge} = \{10, 30, 40\}$
5. $A_1 = \{80\}, A_2 = \{60, 70, 90\}, A_{merge} = \{10, 30, 40, 50\}$
6. $A_1 = \{80\}, A_2 = \{70, 90\}, A_{merge} = \{10, 30, 40, 50, 60\}$
7. $A_1 = \{80\}, A_2 = \{90\}, A_{merge} = \{10, 30, 40, 50, 60, 70\}$
8. $A_1 = \{\}, A_2 = \{90\}, A_{merge} = \{10, 30, 40, 50, 60, 70, 80\}$
9. $A_1 = \{\}, A_2 = \{\}, A_{merge} = \{10, 30, 40, 50, 60, 70, 80, 90\}$

Merge Sort Algorithm

Merge sort works by first recursively sorting each half of the array, and then merging them.

Example

- Originally: $A = \{50, 30, 40, 80, 70, 10, 90, 60\}$.
- Divide A into two halves:
$$A_1 = \{50, 30, 40, 80\}, A_2 = \{70, 10, 90, 60\}.$$
- Sort each half recursively:
$$A_1 = \{30, 40, 50, 80\}, A_2 = \{10, 60, 70, 90\}.$$
- Merge them into the final sorted order:
$$A_{merge} = \{10, 30, 40, 50, 60, 70, 80, 90\}.$$

Pseudo-code of merging

```
algorithm Merge($A_1[1..n_1], A_2[1..n_2]$)
/* merge two sorted arrays */
1. $A_{merge} = \{\}, i = j = 1$
2. while $i \leq n_1$ and $j \leq n_2$
3.     if $A_1[i] \leq A_2[j]$
4.         append $A_1[i]$ to $A_{merge}$, $i = i + 1$
5.     else
6.         append $A_2[j]$ to $A_{merge}$, $j = j + 1$
7. if $i \leq n_1$ then append the unscanned elements of $A_1$ to $A_{merge}$
8. else append the unscanned elements of $A_2$ to $A_{merge}$
9. return $A_{merge}$
```

Pseudo-code of merge sort

```
algorithm MergeSort($A[1..n]$)
1. if $n = 1$ return $A$
2. $A_1 =$ an array containing the first $\lfloor n/2 \rfloor$ numbers of $A$
3. $A_2 =$ an array containing the other elements of $A$
4. $A'_1 =$ MergeSort($A_1$)
5. $A'_2 =$ MergeSort($A_2$)
6. return Merge($A'_1, A'_2$)
```

Worst-case time complexity

Analysis of Merge Sort Algorithm

Let $f(n)$ be the (worst-case) time of sorting an array A of size n. If $n = 1$, $f(n)$ is obviously $O(1)$. Next, we consider $n > 1$. For simplicity, let us assume that n is a power of 2.

- Dividing A into two halves A_1 and A_2 takes $O(n)$ time.
- Sorting A_1 takes at most $f(n/2)$ time.
- Sorting A_2 takes at most $f(n/2)$ time.
- Merging the sorted A_1 and A_2 takes $O(n)$ time. Therefore:

$$f(n) \leq 2f(n/2) + O(n)$$

The next slide proves that $f(n) = O(n \log n)$.

Solving $f(n) \leq 2f(n/2) + O(n)$

$$
\begin{aligned}
f(n) &\leq 2f(n/2) + O(n) \\
&\leq 2(2f(n/4) + O(n/2)) + O(n) = 4f(n/4) + 2O(n) \\
&\leq 4(2f(n/8) + O(n/4)) + 2O(n) = 8f(n/8) + 3O(n) \\
&\dots \\
&\leq n \cdot f(1) + \log n \cdot O(n) \\
&= O(n \log n)
\end{aligned}
$$

To think

Prove $f(n) = O(n \log n)$ without assuming that n is a power of 2.

Playback of this lecture:

The sorting problem.

Bubble sort — worst case $O(n^2)$ time.

Merge sort — worst case $O(n \log n)$ time.

Idea of Quick Sort Algorithm

Select a pivot (partition element).

Rearrange the list so that all the elements before the pivot are smaller than or equal to the pivot and that all the elements after the pivot are than the pivot

Quick Sort： Divide and Conquer

- Quicksort an n-element array:

  --Divide: Partition the array into two subarrays around a pivot x such that elements in lower subarray≤x≤ elements in upper subarray

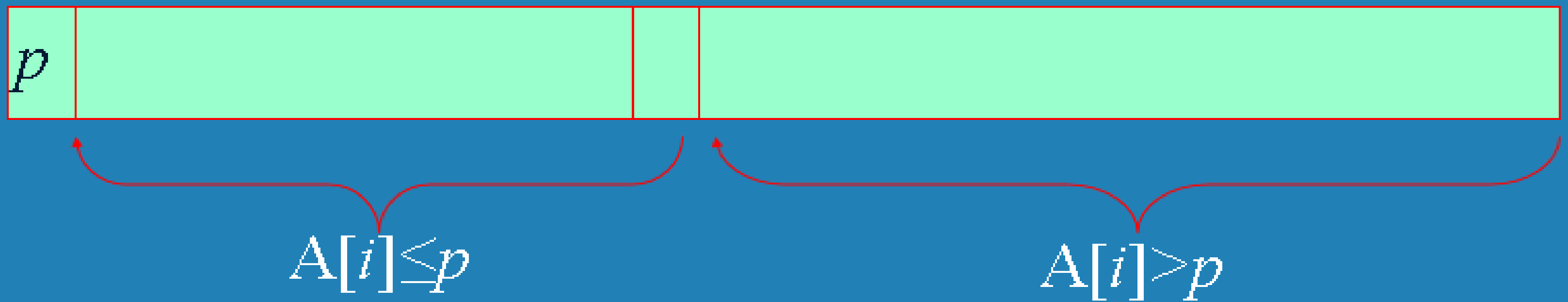
  --Conquer: Recursively sort the subarrays.

  --Combine: Trivial.

Key: Linear-time partitioning subroutine.

Quick sort, an example

- Select a *pivot* (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot (See algorithm *Partition* in section 4.2)
- Exchange the pivot with the last element in the first (i.e., ≤ sublist) – the pivot is now in its final position
- Sort the two sublists



Quick Sort： Divide and Conquer

QUICKSORT

```
QUICKSORT(A, p, r)
1 if p < r
2   then q ← PARTITION( A, p, r)
3          QUICKSORT(A, p, q-1)
4          QUICKSORT(A, q+1, r)
```

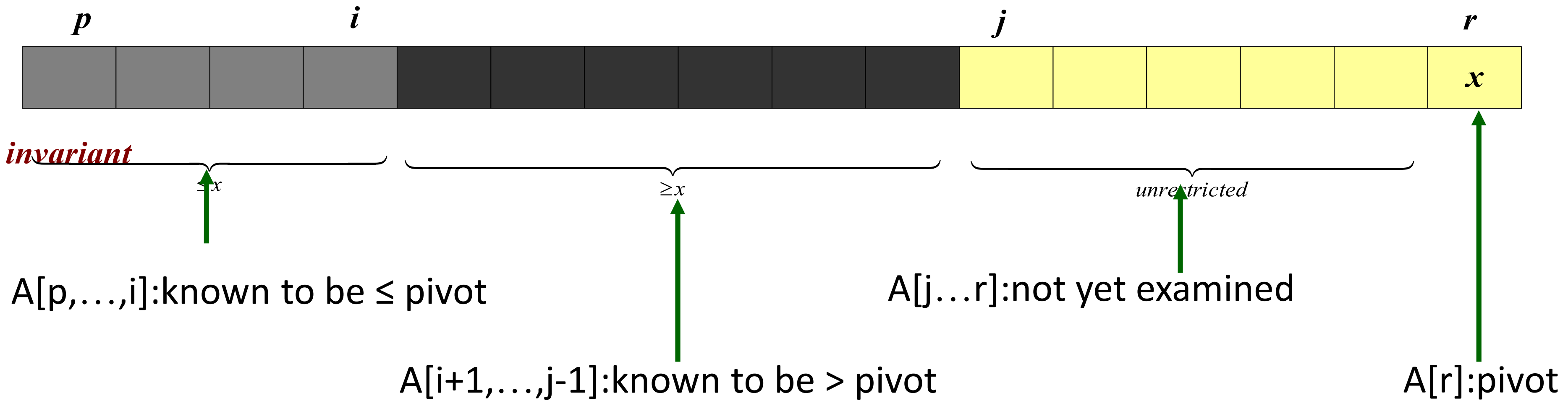
- Initial call Quicksort(A, 1, n)

Partitioning subroutine

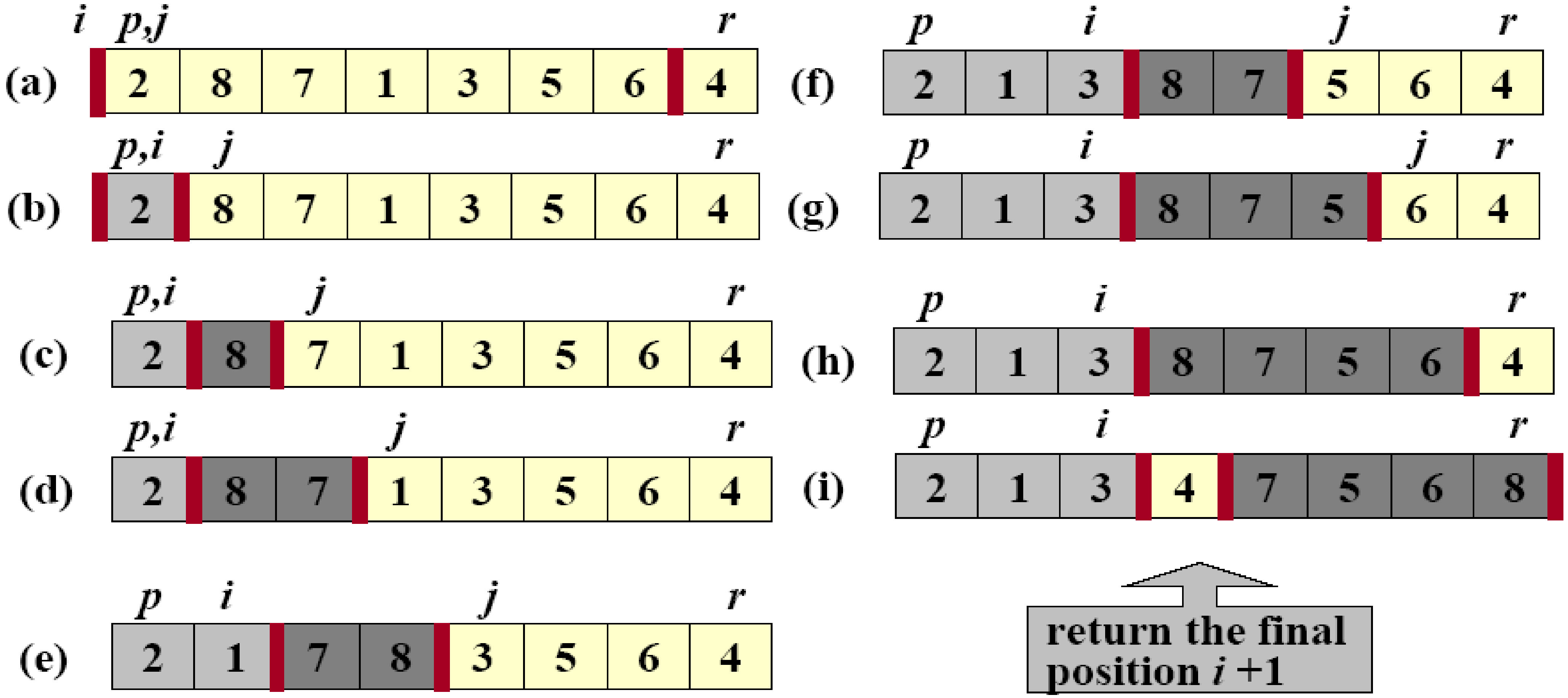
PARTITION

```
PARTITION(A, p, r)          //A[p..r]
1   x ← A[r]                //the rightmost element as pivot
2   i ← p-1
3  for j ← p to r-1
4       do if A[j] ≤ x
5               then i ← i+1
6                    exchange A[i]↔A[j]
7  exchange A[i+1]↔A[r]
8  return i+1
```

Running time = O(n) for n elements



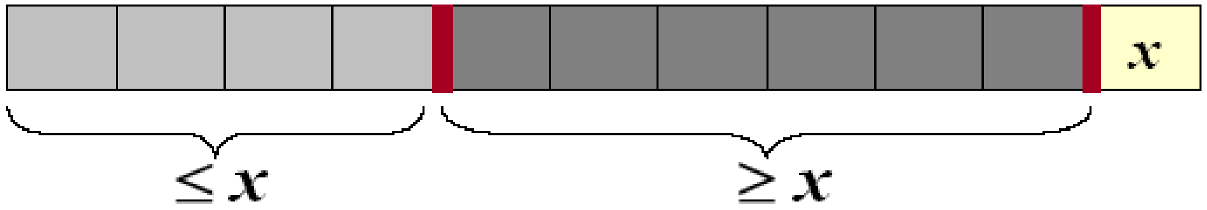
Example of Partitioning


(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8
return the final position i +1

- The operation of Partition on a sample array. Lightly shaded array elements are all with values no greater than x (the pivot). Heavily shaded array elements are all with values greater than x.

Analysis of quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let *T(n) =* worst-case running time on an array of n elements.



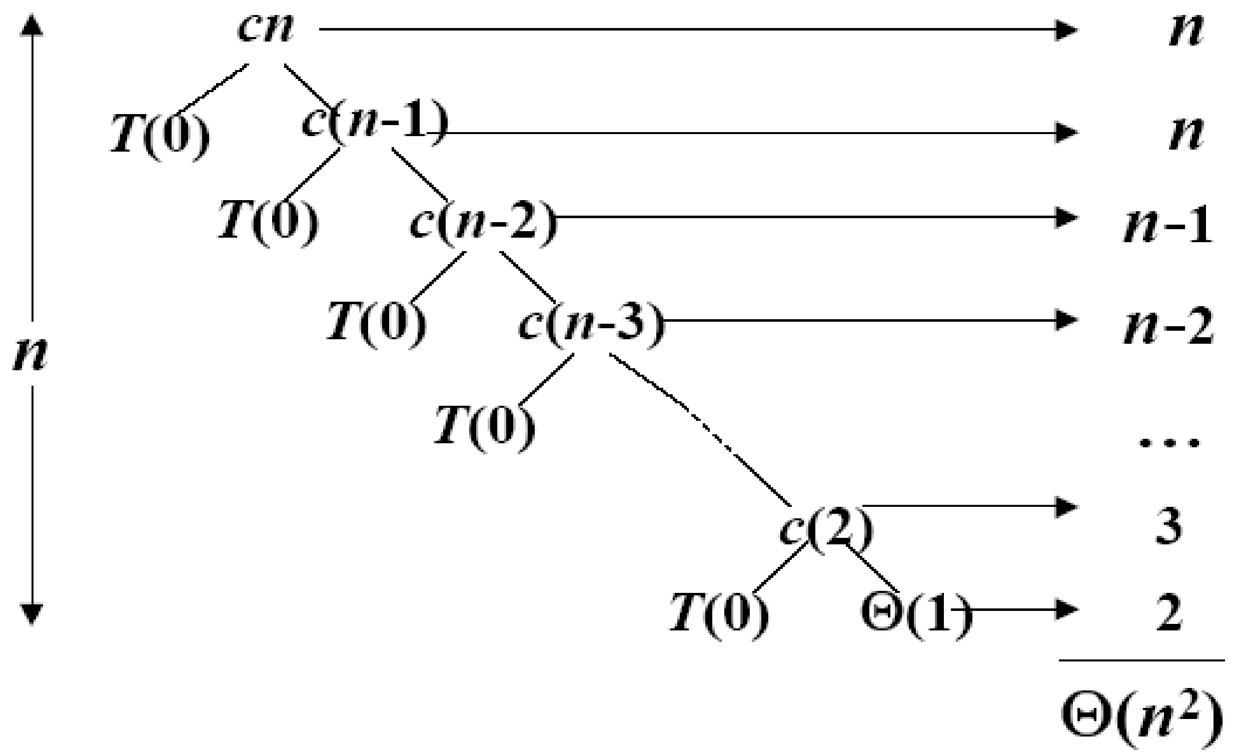
Worst-Case of Quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \text{ (Arithmetic series)} \end{aligned}$$

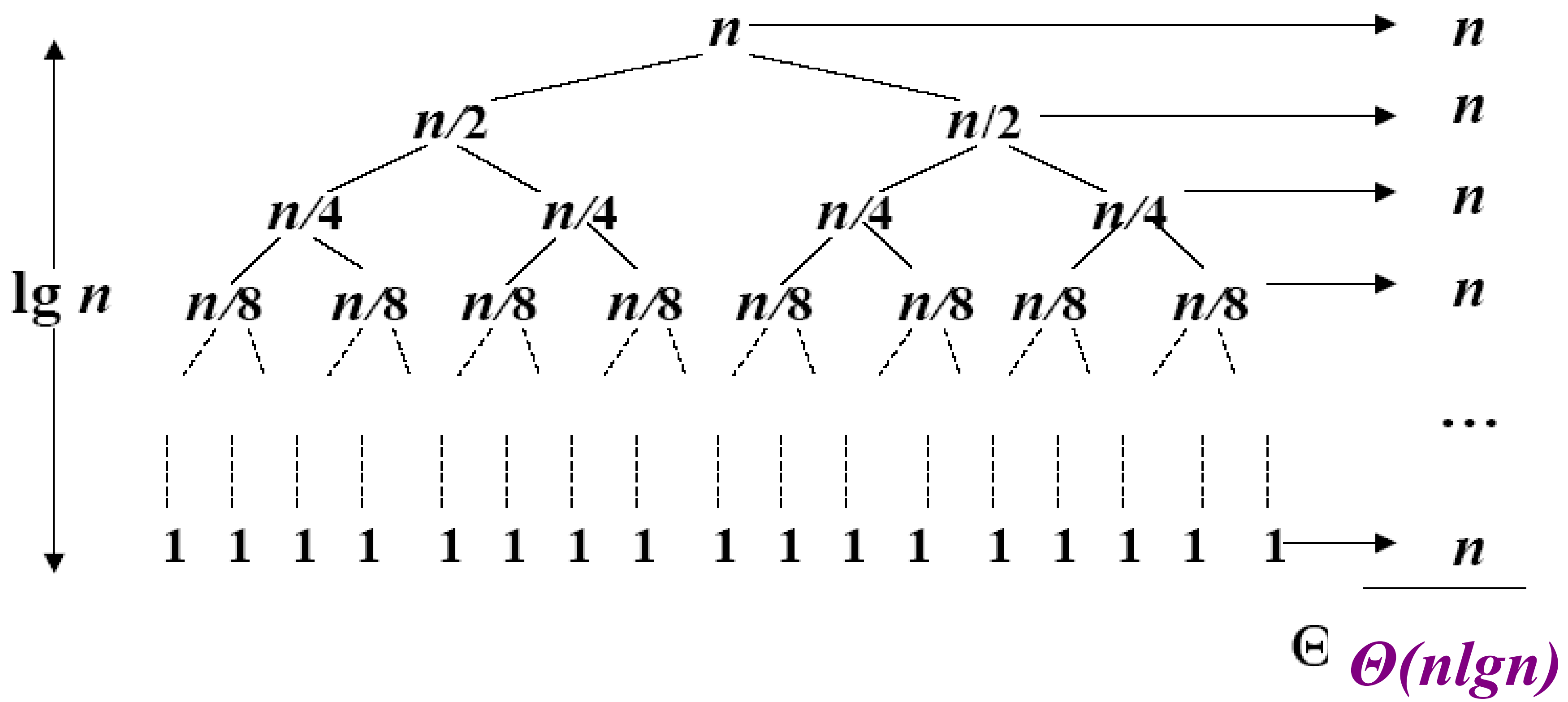
Worst-case recursive tree

- $T(n) = T(n - 1) + T(0) + \Theta(n)$



- A recursion tree for QUICKSORT in which the Partition procedure always puts only a single element on one side of the partition (the worst case). The resulting running time is $\Theta(n^2)$

Best Case Partitioning



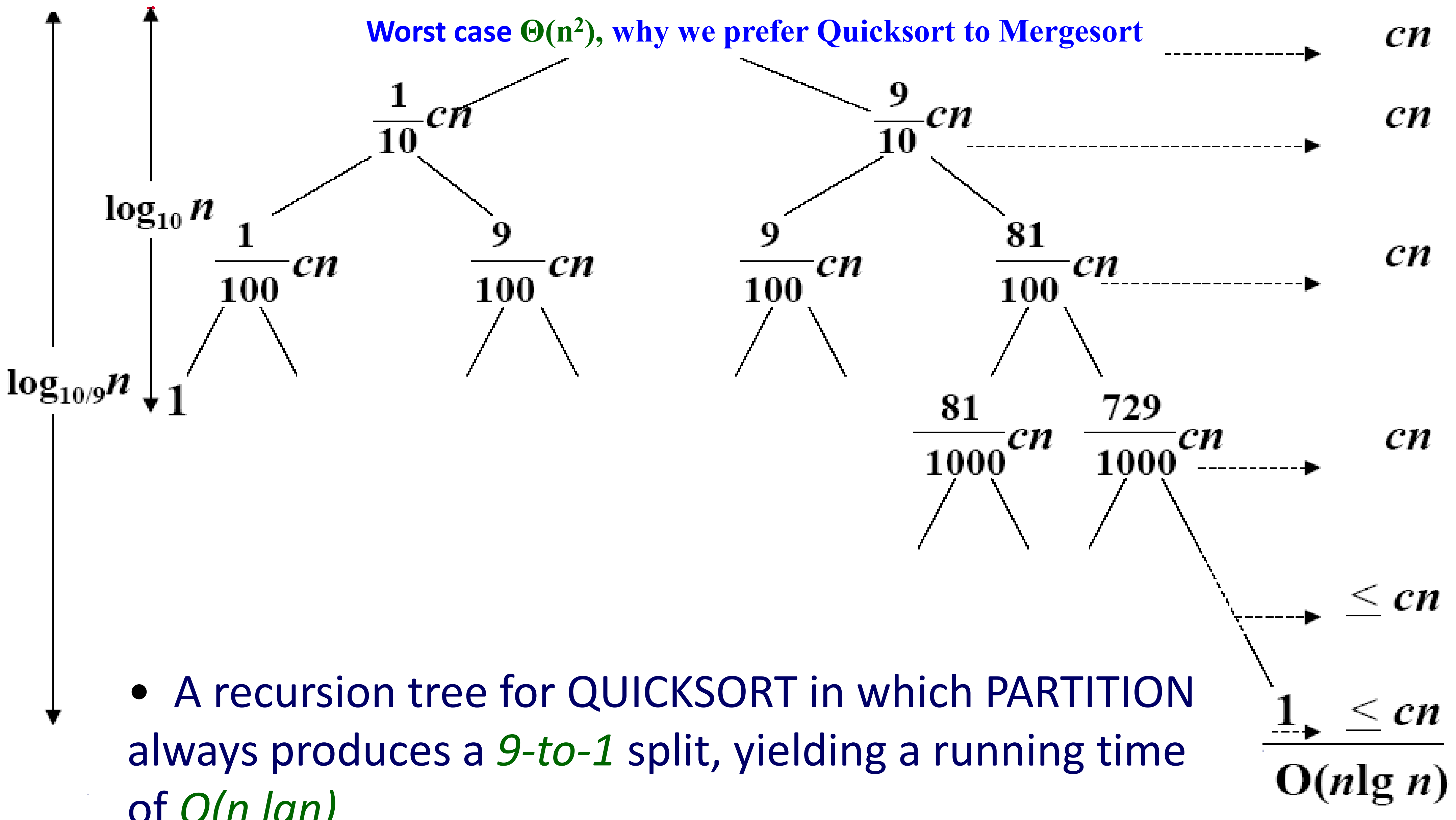
- A recursion tree for QUICKSORT in which the Partition always balances the two sides of the partition equally (the best case). The resulting running time is *Θ(nlgn)*

Best-case analysis

- If we're lucky, PARTITION splits the array evenly:
  --$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\log n)$.

- What if the split is always 1/10:9/10?
  --$T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$
  --What is the solution to this recurrence?

A Recursion tree



- A recursion tree for QUICKSORT in which PARTITION always produces a *9-to-1* split, yielding a running time of *O(n lgn)*

Randomize-Partitioning

RANDOMIZED- PARTITION

```
1   $i \leftarrow$ RANDOM($p$, $r$)
2   exchange $A[r] \leftrightarrow A[i]$
3   return PARTITION($A$, $p$, $r$)
```

RANDOMIZED- QUICKSORT

```
1 if $p < r$
2    then $q \leftarrow$ RANDOMIZED-PARTITION($A$, $p$, $r$)
3         RANDOMIZED-QUICKSORT($A$, $p$, $q-1$)
4         RANDOMIZED-QUICKSORT($A$, $q+1$, $r$)
```

Randomized Quicksort

- Partition around a random element, i.e., around *A[t]*, where *t* chosen uniformly at random from *{p…r}*

- expected time is *O(n lgn)*

Quicksort

- Efficient sorting algorithm

  --Proposed by C.A.R. Hoare in 1962

  --Divide-and-Conquer algorithm

  --Sorts “in place” (like insertion sort, but not like merge sort)

  --Very practical (with tuning)

  --Can be viewed as a randomized Las Vegas algorithm

  --Worst-case running time: $\Theta(n^2)$

  --Expected running time: $\Theta(n\log n)$

  --Constant hidden in $\Theta(n\log n)$ are small.

- Heapsort
  - Heapsort sorts an array by using the heap property we saw earlier

Tutorial:
Implement Selection Sort and Quick Sort in C++

Homework:
Implement Heap Sort
Present the Heap Sort

Heaps

- – A heap is a binary tree with an additional property
- – The value in any node is less than or equal to the value in its parent node. (except for the root node).
- –A heap (or any other binary tree) can be stored in an array

Heap in an array

- – Heap[1] is the root of the tree
- – Heap[2] and Heap[3] are the children of Heap[1]
- – In general, Heap[i] has children Heap[2i] and Heap[2i+1]

- Heapsort

```
Procedure heapsort(T[1..n])

    makeheap(T)

    for i = n to 2 step –1 do
        swap T[1] and T[i]
        siftdown(T[1 .. i – 1], 1)
```

Heapsort

```
procedure makeheap(T[1..n])
        for i = n $\div$ 2 to 1 step –1 do
            siftdown(T, i)
    end for
end
```

Heapsort

T = [7, 2, 9, 5, 1, 3, 8, 4] makeheap
[7, 2, 9, *5*, 1, 3, 8, *4*] siftdown 5 - 0 swaps
[7, 2, *9*, 5, 1, *3*, *8*, 4] siftdown 9 - 0 swaps
[7, *2*, *9*, *5*, 1, 3, 8, 4] siftdown 2 - 2 swaps
[*7*, *5*, *9*, 4, 1, 3, 8, 2] siftdown 7 - 2 swaps
[9, 5, 8, 4, 1, 3, 7, 2] heap complete

n=8 so n/2 = 4 then from 4 to 1 do siftdown.

Heapsort

T = [7, 2, 9, 5, 1, 3, 8, 4] makeheap
[7, 2, 9, *5*, 1, 3, 8, *4*] siftdown 5 - 0 swaps
[7, 2, *9*, 5, 1, *3*, *8*, 4] siftdown 9 - 0 swaps
[7, 2, *9*, *5*, 1, 3, 8, 4] siftdown 2 - 2 swaps
7, *2*, *9*, *5*, 1, 3, 8, 4 > 7, *5*, *9*, 2, 1, 3, 8, 4 > 7, 5, *9*, 4, 1, 3, 8, 2
[*7*, *5*, *9*, 4, 1, 3, 8, 2] siftdown 7 - 2 swaps
[9, 5, 8, 4, 1, 3, 7, 2] heap complete

n=8 so n/2 = 4 then from 4 to 1 do siftdown.

Heapsort : makeheap

T = [7, 2, 9, 5, 1, 3, 8, 4] makeheap
[7, 2, 9, *5*, 1, 3, 8, *4*] siftdown 5 - 0 swaps
[7, 2, *9*, 5, 1, *3*, *8*, 4] siftdown 9 - 0 swaps
[7, *2*, *9*, *5*, 1, 3, 8, 4] siftdown 2 - 2 swaps
[*7*, *5*, *9*, 4, 1, 3, 8, 2] siftdown 7 - 2 swaps
7, *5*, *9*, 4, 1, 3, 8, 2 > *9*, *5*, *7*, 4, 1, 3, 8, 2 > 9, 5, 8, 4, 1, 3, 7, 2
[9, 5, 8, 4, 1, 3, 7, 2] heap complete

n=8 so n/2 = 4 then from 4 to 1 do siftdown.

Heapsort

T = [7, 2, 9, 5, 1, 3, 8, 4]
[9, 5, 8, 4, 1, 3, 7, 2] after makeheap
[2, 5, 8, 4, 1, 3, 7, **9**] swap 2 and 9
[8, 5, 7, 4, 1, 3, 2, **9**] siftdown 2
[2, 5, 7, 4, 1, 3, **8**, **9**] swap 2 and 8
[7, 5, 3, 4, 1, 2, **8**, **9**] siftdown 2
[2, 5, 3, 4, 1, **7**, **8**, **9**] swap 2 and 7
[5, 4, 3, 2, 1, **7**, **8**, **9**] siftdown 2
[1, 4, 3, 2, **5**, **7**, **8**, **9**] swap 1 and 5
[4, 2, 3, 1, **5**, **7**, **8**, **9**] siftdown 1
[1, 2, 3, **4**, **5**, **7**, **8**, **9**] swap 1 and 4
[3, 2, 1, **4**, **5**, **7**, **8**, **9**] siftdown 1
[1, 2, **3**, **4**, **5**, **7**, **8**, **9**] swap 1 and 3
[2, 1, **3**, **4**, **5**, **7**, **8**, **9**] siftdown 1
[1, **2**, **3**, **4**, **5**, **7**, **8**, **9**] swap 1 and 2 - sorted

Heapsort

- Makeheap is in $\Theta(n \log n)$
- Siftdown is in $\Theta(\log n)$
- Heapsort is inΘ $(n \log n) + (n – 1)\ \Theta(\log n) =\Theta(n \log n)$

Simulation

- Applied Algorithms (1)
- Simulation
  - Two basic kinds:
    - Discrete Event (models discrete events)
    - Continuous (models continuous processes)
- Each type is suited to different kinds of simulations
  - Discrete Event – processes where time-based events control what is happening.
    - (e.g. Queues)
  - Continuous – processes where there are no time-based events
    - Physical processes (e.g. Explosions)

- Note: This terminology is not especially obvious or clear.
  - Discrete Event:
    - Usually used to model discrete events
    - Time is continuous, events can occur at any point in time
  - Continuous :
    - Usually used to model a continuous process.
    - Time is broken into discrete chunks (ticks) events only occur on a tick

- Discrete Event Simulation
  - Normally a lot less mathematically complex.
  - Usually requires a lot less computer resources.
- Queue simulation
  - Widely used to evaluate queue-based processes
    - Shops
    - Production lines
    - Industrial processes
  - We will look at some simple examples and see how we might implement them.

- Scenario 1: a single server queue.
  - Customers arrive at random intervals to be served
  - If the server is not busy the customer will be served immediately
  - If the server is busy the customer will join the end of the (possibly empty) queue.
  - When the server has finished with the customer the next customer (if any) begins service – first customer in queue.

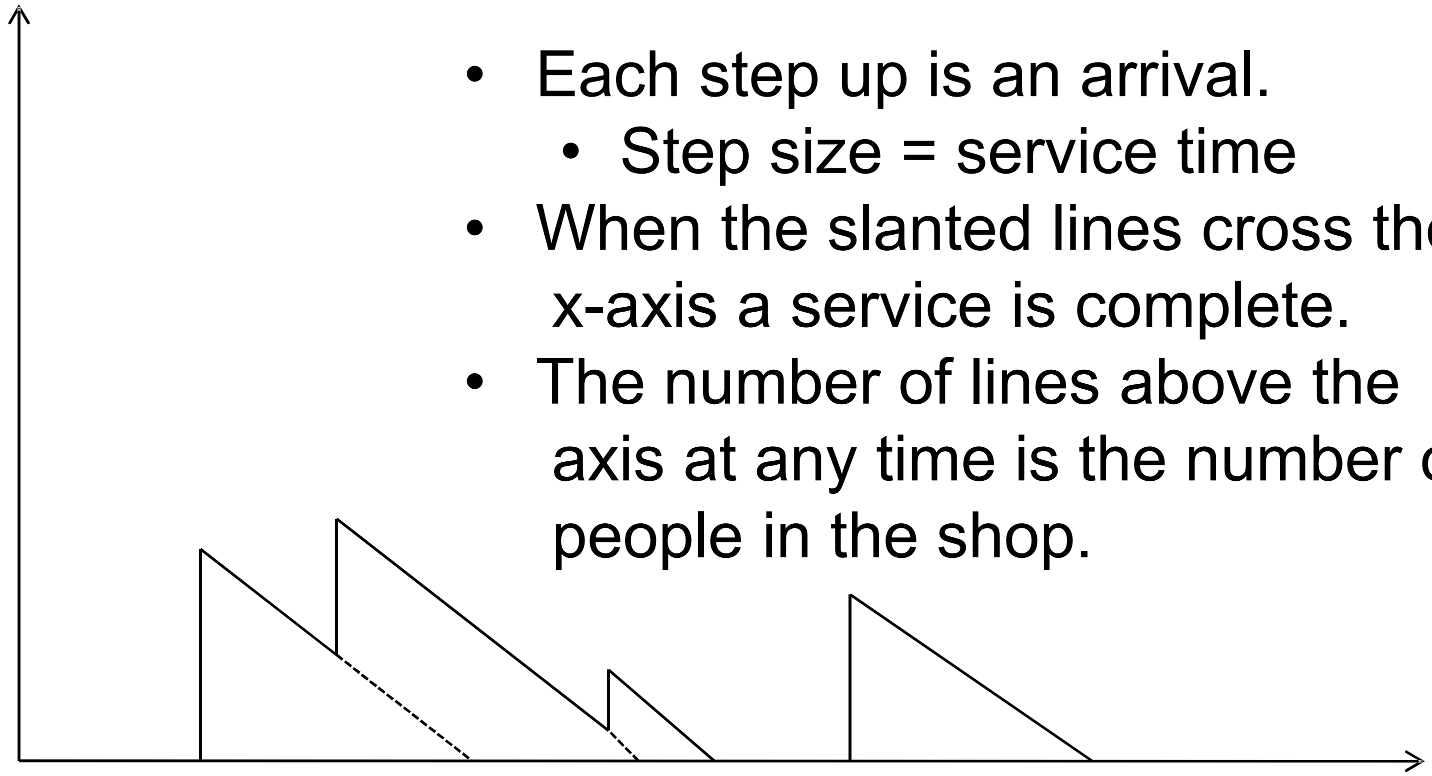
- Scenario 1: Events.
  - Customer arrives
  - Customer starts service
  - Customer ends service and leaves
- What we know:
  - When each customer arrives
  - How long they take to serve
- What we want to know:
  - How big is the queue on average?
  - How busy is the server?
  - What proportion of customers have to wait in a queue?

- Scenario 1: Data
  - Input data is a file consisting of a set of records containing
    - Arrival time
    - Service time
  - For each customer
  - Sorted by arrival time

```
0.24 0.55
0.59 0.16
0.90 0.07
1.87 0.69
 …
```

- Scenario 1: Manual Simulation
  - From the data file we can get a feel for what is happening
  - At time 0.00 the simulation starts
    - The server is idle
    - The queue is empty
  - At time 0.24 the first customer arrives
    - The server is busy for the next 0.55 (until 0.79)
    - The queue is empty
  - At time 0.59 the second customer arrives
    - At time 0.59 the second customer arrives
    - At time 0.59 the second customer arrives
  - At time 0.79 the server finishes with customer 1
    - The server stays busy for the next 0.16 (until 0.95)
    - The queue is empty

- We can represent what is happening with a graph:



- Each step up is an arrival.
  - Step size = service time
- When the slanted lines cross the x-axis a service is complete.
- The number of lines above the axis at any time is the number of people in the shop.

- Starting to design the algorithm.
  - Data structures
    - We need to hold the queue
      - What should we put in it?
    - We need to keep track of the time
    - We need to know if the server is busy
      - If so we need to know when they will finish
    - We need to know when the next customer will arrive
    - We need to track statistics

- Starting to design the algorithm.
  - What does a discrete event simulation look like?

```
repeat
    determine the next event
    process the event
until no more events
```

- Starting to design the algorithm.
  - Procedures
    - Initialise the simulation
    - Process an arrival
    - Process a service completion
    - Finish the simulation
  - Once the simulation is running how do we decide what to do next?
    - Compare the next arrival time with the end of service time

- The main program loop

```
initialise
repeat
    if  busy = true  then
        if  service_end < next arrival  then
            process_service_end
        else
            process_arrival
    else
        process_arrival
    endif
until arrival file is empty and busy = false
finish
```

- Initialise the simulation

```
time = 0
busy = false
queue = empty
read  next_arrival, next_service
```

  - *Set up for statistics collection*

- Process an arrival

```
time = next_arrival
if  busy  then
    enqueue (next_service)
else
    busy = true
    service_end = time + next_service
read (next_arrival next_service)
```

Process a service completion

```
time = service_end

if  queue_empty  then
              busy = false
Else
   service_end = time + dequeue()
```

- What if there is more than one server?
  - Two possible situations
    - One queue for all servers (like a bank)
    - One queue per server (like a supermarket)

- One queue for all servers
  - If all servers are busy add arrival to queue
  - Otherwise make one of the idle servers busy
  - When a server finishes have them serve \
the head of the queue
  - Otherwise make them idle

- The events we are interested in are now:
    - Customer arrives
    - Server 1 finishes
    - Server 2 finishes
    - …
    - Server *n* finishes
- How do we keep track of which event will happen next?
- Do we really need to know which servers are busy or can we just keep track of how many are busy?

- Keeping track of servers:
  - Two possible solutions
    - An array of servers with busy[i] and end_time[i]

      This lets us track who is doing what

      Finding what happens next is in O(*n*)
    - A heap of end times (smallest on top)

      This does not let us track who is doing what

      Finding what happens next is O(log *n*)
    - Can we get the best of both worlds?

- Using a heap
    - If we are clever we can store all event times on the heap
    - All we need is a way to track what each event is

    - One way is to make an event object. This allows us to track the event type and the server id. It doesn't help us track the idle servers, so we could store them in a queue.

    - If we are really clever we can partition the heap into two parts:
        - The heap itself
        - The idle servers
    - How do we manipulate the heap as events occur?

- Initialise the simulation

```
time = 0
n_busy = 0
queue = empty
read heap[0], service_time
```

  - *Set up for statistics collection*

- Customer arrives:

```
    time = heap[0]
    read heap[0], next_service_time
    sift_down(heap, 0)
    if  n_busy < n_servers then
        n_busy = n_busy + 1
        heap[n_busy] = time + service_time
        sift_up(heap, n_busy)
   else
        enqueue(service_time)
    service_time = next_service_time
```

- Server finishes:

```
time = heap[0]
if queue_empty then
    heap[0] = heap[n_busy]
    n_busy = n_busy - 1
else
    heap[0] = time + dequeue()
sift_down(heap)
```

- In summary:
  - Heap Grows if a customer arrives and a server is idle
  - Heap shrinks if a customer is served and the queue is empty
  - Heap stays the same size otherwise
- If we keep a second array (id) initially filled with integers 0..*n* we can use it to track who is doing what
  - 0 is the next arrival
  - 1 is server 1's completion time
  - 2 is server 2's completion time
  - …
  - *n* is server *n*'s completion time

- Customer arrives becomes:

```
time = heap[0]
read heap[0], next_service_time
sift_down(heap, id, 0)
if  n_busy < n_servers then
    n_busy = n_busy + 1
    heap[n_busy] = time + service_time
    sift_up(heap, id, n_busy)
    service_time = next_service_time
else
    enqueue(next_service_time)
```

- Server finishes:

```
    time = heap[0]
    if queue_empty then
        swap (id[0], id[n_busy])
        heap[0] = heap[n_busy]
        n_busy = n_busy - 1
    else
        heap[0] = time + dequeue()
    sift_down(heap, id, 0)
```

```
repeat
        determine the next event
        process the event
 until no more events
```

- If the top of the id array is a zero the next event is an arrival
- If the top of the id array is non_zero the next event is a service completion for server id[0]

- Every time we move an entry in the heap we move the corresponding entry in the id array.
- The simulation starts with the first arrival time in heap[0] and n_busy = 0

- Multiple queues
  - In this case we have an array of *n* queues, 1 per server
  - When a customer arrives and all servers are busy we place the customer on one of the queues (which one?)
  - When a server finishes we only make them busy if their queue is not empty
  - NOTE: This means that we can have a queue even if a server is idle.

- Priority queues
  - How would we handle the situation where customers are given different service priorities?
    - One queue for each priority empty the highest priority queue first
    - This is only efficient if there are a small number of priorities
  - What do we do if each priority may be different?
    - E.g. priority is a float between 0 and 1
    - 0 is the lowest customer priority
    - 1 is the highest priority customer
    - We have an infinite number of different priorities so we can’t have a queue for each one.

- Priority queues
  - The solution is to replace the queue with a heap ordered on priority
  - Each time we remove a customer from the heap we
    - Move the last entry to the top of the heap
    - Reduce the heap size by one
    - Sift down the top entry
  - Each time we add a customer to the heap we:
    - Increase the heap size by one
    - Add the customer to the end of the heap
    - Sift up the last entry

- Continuous Simulation.
  - Often dependent on complex mathematics.
  - Often based on gridded algorithms.
    - Implicit solution
    - Explicit solution
  - Often requires extreme computing resources.
    - Supercomputers
  - We will not be looking at continuous simulation in this subject but here are a couple of examples to help you understand the differences.

- Examples of continuous simulation: 1
  - Mine explosions
    - Solve 10 differential equations for each of several thousand cells for every millisecond of the simulated event.
    - Parallel supercomputer.
    - Still over 24 hours per run.
  - Some videos of the results…

- Examples of continuous simulation: 2
  - Social simulation
    - Track the state of individuals in a simulated environment.
  - Two entities
    - Peeps – simulated individuals (not always people)
    - Cells – simulated locations
  - Used to examine response to threat
    - Spread of disease
    - Natural disaster
    - Man-made disaster (terrorist attack)
  - Again, lots of time/computer power required.

Pseudo code

Quick Sort Algorithm

```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //          indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ← Partition(A[l..r])  //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```