CSCI 4470/6470 Algorithms, Spring 2017

Lecture Note II

Liming Cai, Department of Computer Science, UGA

January 24, 2017

Part II Sorting and Order Statistics

Part II Sorting and Order Statistics

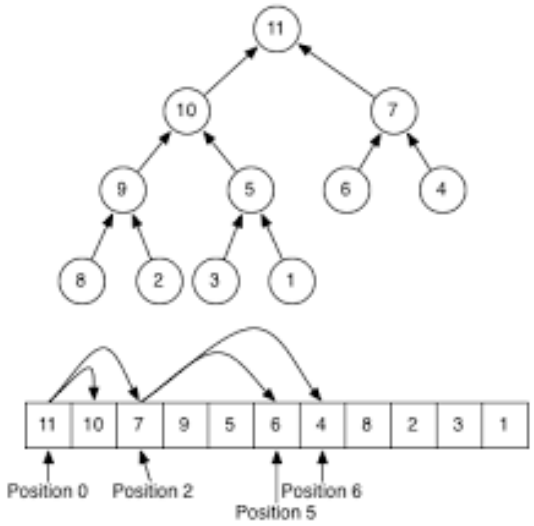
- Chapter 6. Heapsort, the use of priority queue
- Chapter 7. Quicksort, probabilistic analysis, randomized algorithms
- Chapter 8. Sorting in linear time, lower bounds
- Chapter 9. Medians and order statistics

Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue

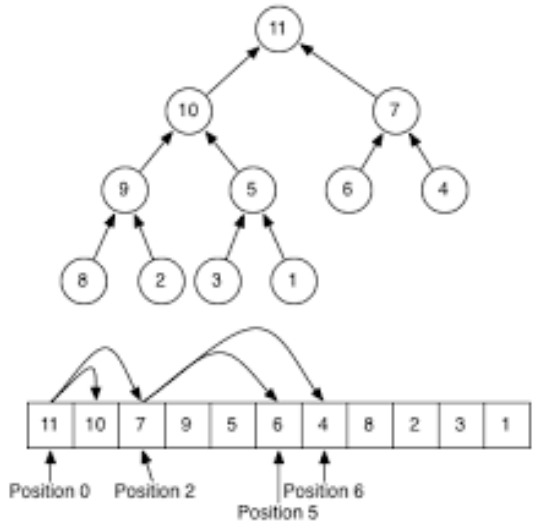
Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue



Chapter 6. Heapsort

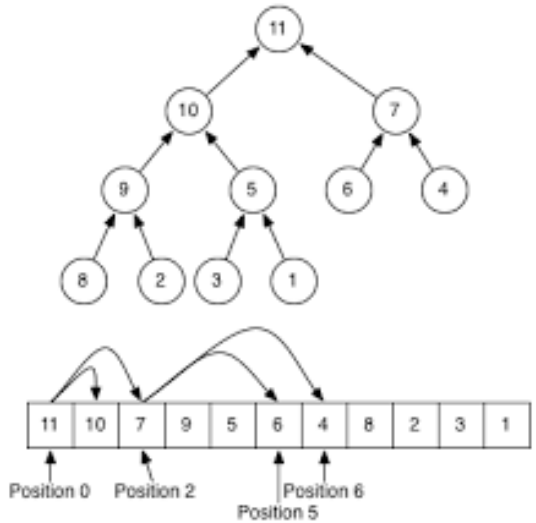
Chapter 6. Heapsort and the use of priority queue



- key(parent) $\geq$ key(leftChild), key(rightChild);

Chapter 6. Heapsort

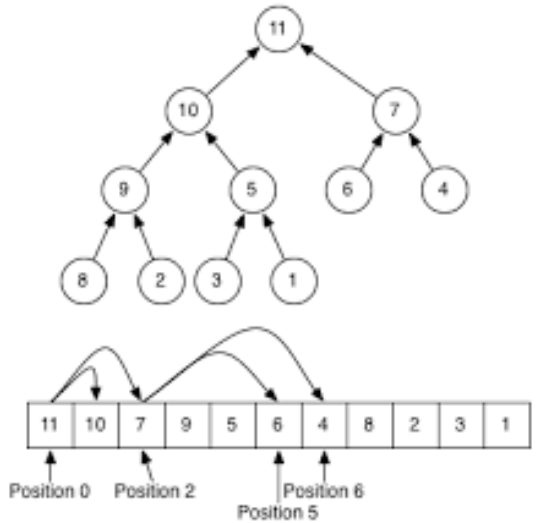
Chapter 6. Heapsort and the use of priority queue



- key(parent) $\geq$ key(leftChild), key(rightChild);
- relationships are modeled with a complete binary tree

Chapter 6. Heapsort

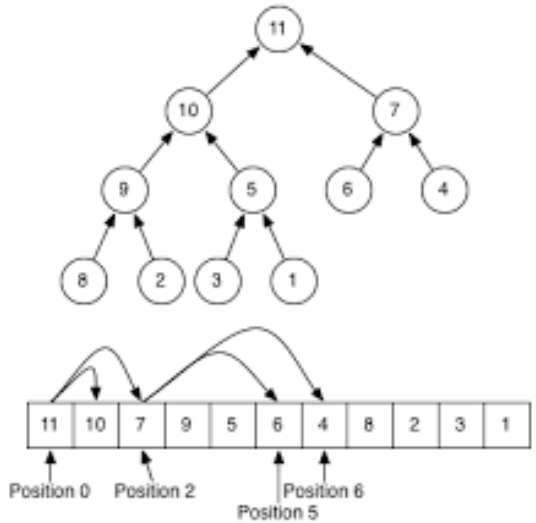
Chapter 6. Heapsort and the use of priority queue



- key(parent) $\geq$ key(leftChild), key(rightChild);
- relationships are modeled with a complete binary tree
- can be stored in arrays (indexes begin with 0),

Chapter 6. Heapsort

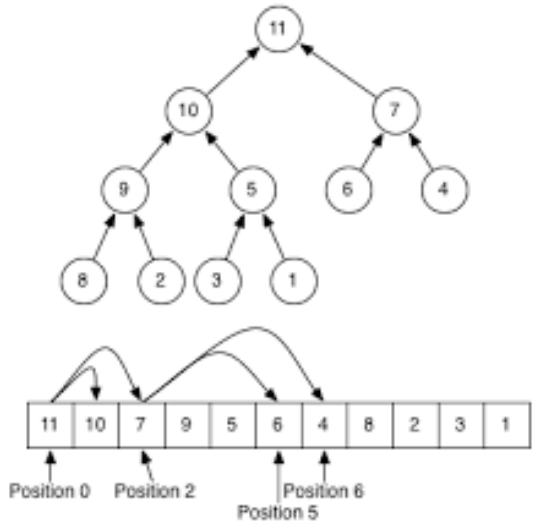
Chapter 6. Heapsort and the use of priority queue



- key(parent) $\geq$ key(leftChild), key(rightChild);
- relationships are modeled with a complete binary tree
- can be stored in arrays (indexes begin with 0), index(leftChild) $= 2\times$ index(parent) $+1$

Chapter 6. Heapsort

Chapter 6. Heapsort and the use of priority queue



- key(parent) $\geq$ key(leftChild), key(rightChild);
- relationships are modeled with a complete binary tree
- can be stored in arrays (indexes begin with 0),
  index(leftChild) $= 2\times$ index(parent) $+1$
  index(rightChild) $= 2\times$ index(parent) $+2$

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- BUILD-MAX-HEAP(A)

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- BUILD-MAX-HEAP(A)
- MAX-HEAPIFY(A, i)

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- Build-Max-Heap(A)
- Max-Heapify(A, i)
- HeapSort(A)

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- BUILD-MAX-HEAP(A)
- MAX-HEAPIFY(A, i)
- HEAPSORT(A)

heaps as priority queues

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- Build-Max-Heap(A)
- Max-Heapify(A, i)
- HeapSort(A)

heaps as priority queues

- Heap-Maximum(A)

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- Build-Max-Heap(A)
- Max-Heapify(A, i)
- HeapSort(A)

heaps as priority queues

- Heap-Maximum(A)
- Heap-Extract-Max(A)

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- Build-Max-Heap(A)
- Max-Heapify(A, i)
- HeapSort(A)

heaps as priority queues

- Heap-Maximum(A)
- Heap-Extract-Max(A)
- Heap-Increase-Key(A, I, key)

Chapter 6. Heapsort

The heap sort algorithm consists of subroutines:

- Build-Max-Heap(A)
- Max-Heapify(A, i)
- HeapSort(A)

heaps as priority queues

- Heap-Maximum(A)
- Heap-Extract-Max(A)
- Heap-Increase-Key(A, I, key)
- Max-Heap-Insert(A, key)

Chapter 6. Heapsort

Algorithm HeapSort(A)

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = length[A]$ **downto** 2

Chapter 6. Heapsort

Algorithm HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$

Chapter 6. Heapsort

Algorithm HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$
4. $heapsize[A] = heapsize[A] - 1$

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$
4. $heapsize[A] = heapsize[A] - 1$
5. Max-Heapify($A, 1$)

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$
4. $heapsize[A] = heapsize[A] - 1$
5. Max-Heapify($A, 1$)

$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n, 1)$

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$
4. $heapsize[A] = heapsize[A] - 1$
5. Max-Heapify($A, 1$)

$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n, 1)$

Subroutine Build-Max-Heap(A)

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$
4. $heapsize[A] = heapsize[A] - 1$
5. Max-Heapify($A, 1$)

$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n, 1)$

Subroutine Build-Max-Heap(A)

1. $heapsize[A] = length[A]$

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$
4. $heapsize[A] = heapsize[A] - 1$
5. Max-Heapify$(A, 1)$

$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n, 1)$

Subroutine Build-Max-Heap(A)

1. $heapsize[A] = length[A]$
2. **for** $i = \lfloor \frac{1}{2} length[A] \rfloor$ **downto** 1

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$
4. $heapsize[A] = heapsize[A] - 1$
5. Max-Heapify$(A, 1)$

$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n, 1)$

Subroutine Build-Max-Heap(A)

1. $heapsize[A] = length[A]$
2. **for** $i = \lfloor \frac{1}{2} length[A] \rfloor$ **downto** 1
3. Max-Heapify(A, i)

Chapter 6. Heapsort

Algorithm HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = length[A]$ **downto** 2
3. exchange $A[1] \longleftrightarrow A[i]$
4. $heapsize[A] = heapsize[A] - 1$
5. Max-Heapify$(A, 1)$

$$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n,1)$$

Subroutine Build-Max-Heap(A)

1. $heapsize[A] = length[A]$
2. **for** $i = \lfloor \frac{1}{2} length[A] \rfloor$ **downto** 1
3. Max-Heapify(A, i)

$$T_{BMH}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} T_{MH}(n,i)$$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$
10. $\textsc{Max-Heapify}(A, largest)$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$
10. $\textsc{Max-Heapify}(A, largest)$

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$
10. $\textsc{Max-Heapify}(A, largest)$

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

$T_{MH}(n, i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$
10. $\textsc{Max-Heapify}(A, largest)$

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

$T_{MH}(n, i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

$$T_{BMH}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} T_{MH}(n, i)$$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$
10. $\textsc{Max-Heapify}(A, largest)$

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

$T_{MH}(n, i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

$$T_{BMH}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} T_{MH}(n, i) \leq c \frac{n}{2} \log_2 n$$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$
10. $\textsc{Max-Heapify}(A, largest)$

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

$T_{MH}(n, i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

$$T_{BMH}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} T_{MH}(n, i) \leq c\frac{n}{2} \log_2 n$$

$$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n, 1)$$

Chapter 6. Heapsort

Subroutine $\textsc{Max-Heapify}(A, i)$

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$
10. $\textsc{Max-Heapify}(A, largest)$

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

$T_{MH}(n, i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

$$T_{BMH}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} T_{MH}(n, i) \leq c\frac{n}{2} \log_2 n$$

$$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n, 1) \leq c\frac{n}{2} \log_2 n + (n-1)c \log_2 n$$

Chapter 6. Heapsort

Subroutine MAX-HEAPIFY(A, i)

1. $l = LEFT[i]$
2. $r = RIGHT[i]$
3. **if** $(l \leq heapsize[A])$ **and** $(A[l] > A[i])$
4. **then** $largest = l$
5. **else** $largest = i$
6. **if** $(r \leq heapsize[A])$ **and** $(A[r] > A[largest])$
7. **then** $largest = r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \longleftrightarrow A[largest]$
10. MAX-HEAPIFY$(A, largest)$

$T_{MH}(n, i) = c + T_{MH}(n, 2i)$

$T_{MH}(n, i) \leq c \log_2 n$, for all $i = 1, 2, \ldots, n$.

$$T_{BMH}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} T_{MH}(n, i) \leq c\frac{n}{2} \log_2 n$$

$$T_{HS}(n) = T_{BMH}(n) + (n-1)T_{MH}(n, 1) \leq c\frac{n}{2} \log_2 n + (n-1)c \log_2 n \leq O(n \log n)$$

Chapter 6. Heapsort

Operations on heaps:

Chapter 6. Heapsort

Operations on heaps:

Function HEAP-MAXIMUM(A) — obtain the maximum

1. **return** ($A[1]$)

Chapter 6. Heapsort

Operations on heaps:

Function HEAP-MAXIMUM(A) — obtain the maximum

1. **return** ($A[1]$)

Function HEAP-EXTRACT-MAX(A) — obtain and remove the maximum

1. **if** $heapsize[A] < 1$
2. **then return** ("heap underflow")
3. $max = A[1]$
4. $A[1] = A[heapsize[A]]$
5. $heapsize[A] = heapsize[A] - 1$
6. MAX-HEAPIFY($A, 1$)
7. **return** (max)

Chapter 6. Heapsort

Operations on heaps:

Function HEAP-MAXIMUM(A) — obtain the maximum

1. **return** ($A[1]$)

Function HEAP-EXTRACT-MAX(A) — obtain and remove the maximum

1. **if** $heapsize[A] < 1$
2. **then return** ("heap underflow")
3. $max = A[1]$
4. $A[1] = A[heapsize[A]]$
5. $heapsize[A] = heapsize[A] - 1$
6. MAX-HEAPIFY($A, 1$)
7. **return** (max)

Function HEAP-INCREASE-KEY(A, i, key) — replace a key with a larger value

1. **if** $key < A[i]$
2. **then return** ("new key is smaller than current key")
3. $A[i] = key$
4. **while** $i > 1$ **and** $A[PARENT[i]] < A[i]$
5. exchange $A[i] \longleftrightarrow A[PARENT[i]]$
6. $i = PARENT[i]$

Chapter 6. Heapsort

Operations on heaps:

Function HEAP-MAXIMUM(A) — obtain the maximum

1. **return** ($A[1]$)

Function HEAP-EXTRACT-MAX(A) — obtain and remove the maximum

1. **if** $heapsize[A] < 1$
2. **then return** ("heap underflow")
3. $max = A[1]$
4. $A[1] = A[heapsize[A]]$
5. $heapsize[A] = heapsize[A] - 1$
6. MAX-HEAPIFY($A, 1$)
7. **return** (max)

Function HEAP-INCREASE-KEY(A, i, key) — replace a key with a larger value

1. **if** $key < A[i]$
2. **then return** ("new key is smaller than current key")
3. $A[i] = key$
4. **while** $i > 1$ **and** $A[PARENT[i]] < A[i]$
5. exchange $A[i] \longleftrightarrow A[PARENT[i]]$
6. $i = PARENT[i]$

Function MAX-HEAP-INSERT(A, key) — insert a new key to heap

1. $heapsize[A] = heapsize[A] + 1$
2. $A[heapsize[A]] = -\infty$
3. HEAP-INCREASE-KEY($A, heapsize[A], key$)

Chapter 7. Quicksort and randomized algorithms

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

- divide: re-organize list $A[p, r]$ into two sublists $A[p, q-1]$ and $A[q+1, r]$ based on pivot $A[q]$, such that

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

- divide: re-organize list $A[p, r]$ into two sublists $A[p, q-1]$ and $A[q+1, r]$ based on pivot $A[q]$, such that

  (a) $A[i] \leq A[q]$ for all $i = p, \cdots, q-1$

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

- divide: re-organize list $A[p, r]$ into two sublists $A[p, q-1]$ and $A[q+1, r]$ based on pivot $A[q]$, such that

  (a) $A[i] \leq A[q]$ for all $i = p, \cdots, q-1$
  (b) $A[i] \geq A[q]$ for all $i = q+1, \cdots, r$

Chapter 7. Quicksort and randomized algorithms

Idea of the Quicksort: divide-and-conquer

- divide: re-organize list $A[p, r]$ into two sublists $A[p, q-1]$ and $A[q+1, r]$ based on pivot $A[q]$, such that

  (a) $A[i] \leq A[q]$ for all $i = p, \cdots, q-1$
  (b) $A[i] \geq A[q]$ for all $i = q+1, \cdots, r$

- conquer: sort $A[p, q-1]$ and $A[q+1, r]$, recursively.

Chapter 7. Quicksort and Randomized algorithms

Chapter 7. Quicksort and Randomized algorithms

Algorithm $\textsc{QuickSort}\ (A, p, r)$

Chapter 7. Quicksort and Randomized algorithms

Algorithm QUICKSORT (A, p, r)

1. **if** $p < r$

Chapter 7. Quicksort and Randomized algorithms

Algorithm $\textsc{QuickSort}\ (A, p, r)$

1. **if** $p < r$
2. **then** $q = \textsc{Partition}(A, p, r)$

Chapter 7. Quicksort and Randomized algorithms

Algorithm QUICKSORT (A, p, r)

1. **if** $p < r$
2. **then** $q =$ PARTITION(A, p, r)
3. QUICKSORT $(A, p, q-1)$

Chapter 7. Quicksort and Randomized algorithms

Algorithm $\textsc{QuickSort}\ (A, p, r)$

1. **if** $p < r$
2. **then** $q = \textsc{Partition}(A, p, r)$
3. $\textsc{QuickSort}\ (A, p, q-1)$
4. $\textsc{QuickSort}\ (A, q+1, r)$

Chapter 7. Quicksort and Randomized algorithms

Algorithm $\textsc{QuickSort}\ (A, p, r)$

1. **if** $p < r$
2. **then** $q = \textsc{Partition}(A, p, r)$
3. $\textsc{QuickSort}\ (A, p, q-1)$
4. $\textsc{QuickSort}\ (A, q+1, r)$

How the pivot $A[q]$ is identified is crucial to the performance of Quicksort.

Chapter 7. Quicksort and Randomized algorithms

Algorithm $\textsc{QuickSort}\ (A, p, r)$
1. **if** $p < r$
2. **then** $q = \textsc{Partition}(A, p, r)$
3. $\textsc{QuickSort}\ (A, p, q-1)$
4. $\textsc{QuickSort}\ (A, q+1, r)$

How the pivot $A[q]$ is identified is crucial to the performance of Quicksort.

- Assume $A[q]$ partitions list A, p, r evenly, then $T(n) \leq 2T(n/2) + cn = O(n \log_2 n)$

Chapter 7. Quicksort and Randomized algorithms

Algorithm QUICKSORT (A, p, r)

1. **if** $p < r$
2. **then** $q =$ PARTITION(A, p, r)
3. QUICKSORT $(A, p, q-1)$
4. QUICKSORT $(A, q+1, r)$

How the pivot $A[q]$ is identified is crucial to the performance of Quicksort.

- Assume $A[q]$ partitions list A, p, r evenly, then $T(n) \leq 2T(n/2) + cn = O(n \log_2 n)$
- Assume $A[q]$ partitions the list 20% vs 80%, then $T(n) \leq T(\frac{n}{5}) + T(\frac{4n}{5}) + cn = O(n \log_2 n)$

Chapter 7. Quicksort and Randomized algorithms

Algorithm $\textsc{QuickSort}\ (A, p, r)$

1. **if** $p < r$
2. **then** $q = \textsc{Partition}(A, p, r)$
3. $\textsc{QuickSort}\ (A, p, q-1)$
4. $\textsc{QuickSort}\ (A, q+1, r)$

How the pivot $A[q]$ is identified is crucial to the performance of Quicksort.

- Assume $A[q]$ partitions list A, p, r evenly, then $T(n) \leq 2T(n/2) + cn = O(n \log_2 n)$
- Assume $A[q]$ partitions the list 20% vs 80%, then $T(n) \leq T(\frac{n}{5}) + T(\frac{4n}{5}) + cn = O(n \log_2 n)$
- Assume $A[q]$ partitions the list 1% vs 99%, then $T(n) \leq T(\frac{n}{100}) + T(\frac{99n}{100}) + cn = O(n \log_2 n)$

Chapter 7. Quicksort and Randomized algorithms

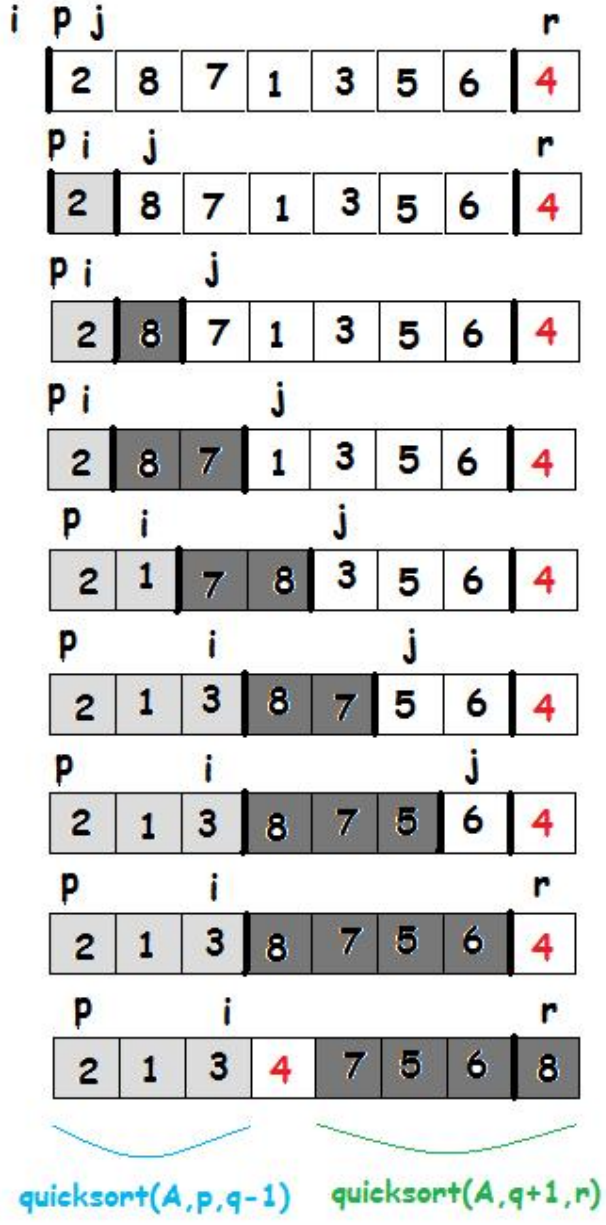
Algorithm $\textsc{QuickSort}\ (A, p, r)$
1. **if** $p < r$
2. **then** $q = \textsc{Partition}(A, p, r)$
3. $\textsc{QuickSort}\ (A, p, q-1)$
4. $\textsc{QuickSort}\ (A, q+1, r)$

How the pivot $A[q]$ is identified is crucial to the performance of Quicksort.

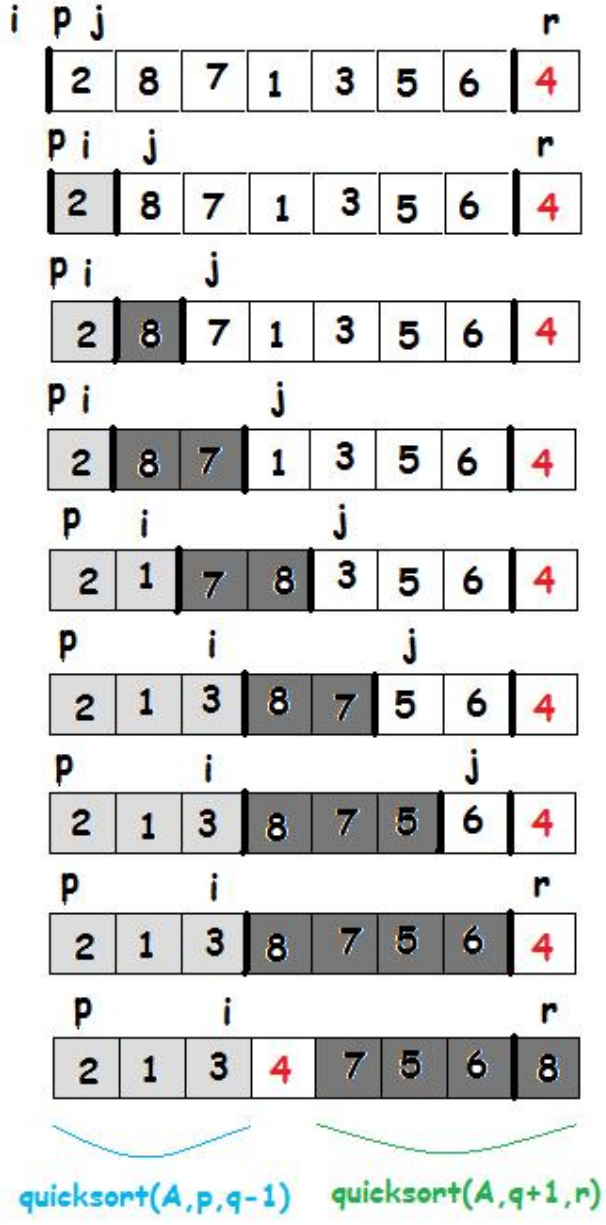
- Assume $A[q]$ partitions list A, p, r evenly, then $T(n) \leq 2T(n/2) + cn = O(n \log_2 n)$
- Assume $A[q]$ partitions the list 20% vs 80%, then $T(n) \leq T(\frac{n}{5}) + T(\frac{4n}{5}) + cn = O(n \log_2 n)$
- Assume $A[q]$ partitions the list 1% vs 99%, then $T(n) \leq T(\frac{n}{100}) + T(\frac{99n}{100}) + cn = O(n \log_2 n)$

How can we identify such a pivot?

Chapter 7. Quicksort



Chapter 7. Quicksort



PARTITION(A, p, r)

```
1  x ← A[r]
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
```

Chapter 7. Quicksort

PARTITION may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$.

Chapter 7. Quicksort

PARTITION may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$.

- skewed situation like $1 : n - 1$ partition may happen, resulting in running time $\geq cn^2$.

Chapter 7. Quicksort

PARTITION may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$.

- skewed situation like $1 : n - 1$ partition may happen, resulting in running time $\geq cn^2$.
- however, chances for skewed cases like above are very small.

Chapter 7. Quicksort

PARTITION may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1 - \epsilon)n$, for a constant $\epsilon > 0$.

- skewed situation like $1 : n - 1$ partition may happen, resulting in running time $\geq cn^2$.
- however, chances for skewed cases like above are very small.
- that is, the cases other than the skewed ones occur much more often.

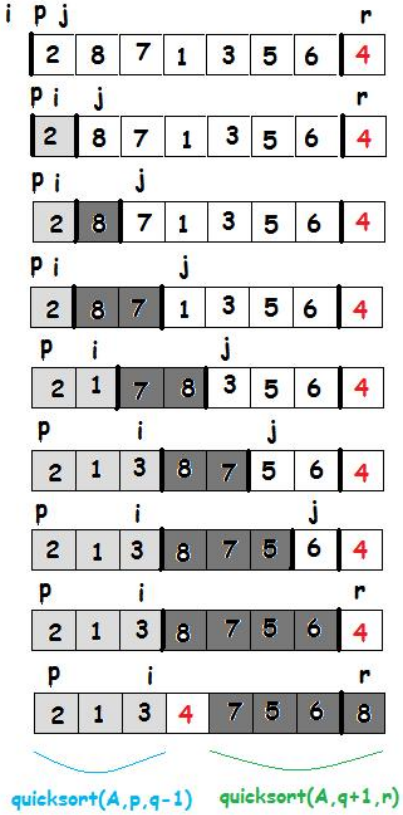
Chapter 7. Quicksort

PARTITION may not guarantee to partition the list to two fractions of sizes $\epsilon n : (1-\epsilon)n$, for a constant $\epsilon > 0$.

- skewed situation like $1 : n-1$ partition may happen, resulting in running time $\geq cn^2$.
- however, chances for skewed cases like above are very small.
- that is, the cases other than the skewed ones occur much more often.

So the idea of Quicksort may work well on a majority of data.

Chapter 7. Quicksort

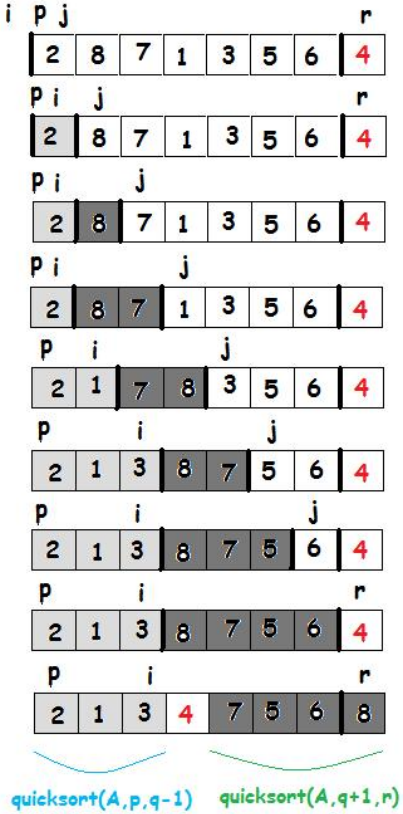


Assume that the equal likely chance for every number to be in the last position, what is the chance to partition the list into

$$x\% \text{ vs } (100-x)\%$$

fragments, for $10 \leq x \leq 90$?

Chapter 7. Quicksort



Assume that the equal likely chance for every number to be in the last position, what is the chance to partition the list into

$$x\% \text{ vs } (100-x)\%$$

fragments, for $10 \leq x \leq 90$?

The chance is $= 80\%$

What running time would it be if 10:90 partition is always guaranteed?

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

l_0: cn cn

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

l_0:		cn		cn
l_1:	$cn/10$		$9cn/10$	cn

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

l_0:			cn		cn
l_1:		$cn/10$		$9cn/10$	cn
l_2:	$cn/10^2$	$9cn/10^2$	$9cn/10^2$	$9^2cn/10^2$	cn

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lccccc}
l_0: & & & cn & & cn \\
l_1: & & cn/10 & & 9cn/10 & cn \\
l_2: & cn/10^2 & 9cn/10^2 & 9cn/10^2 & 9^2cn/10^2 & cn \\
 & & & \cdots\cdots & &
\end{array}$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lccccccc}
l_0: & & & cn & & & & cn \\
l_1: & & cn/10 & & & 9cn/10 & & cn \\
l_2: & cn/10^2 & & 9cn/10^2 & 9cn/10^2 & & 9^2cn/10^2 & cn \\
 & & & \cdots\cdots & & & & \\
l_h: & cn/10^h & & & & & c9^h n/10^h & cn
\end{array}$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$
\begin{array}{lccccccc}
l_0: & & & cn & & & & cn \\
l_1: & & cn/10 & & & 9cn/10 & & cn \\
l_2: & cn/10^2 & & 9cn/10^2 & 9cn/10^2 & & 9^2cn/10^2 & cn \\
 & & & \cdots\cdots & & & & \\
l_h: & cn/10^h & & & & & c9^h n/10^h & cn \\
 & & & \cdots\cdots & & & &
\end{array}
$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lccccccc}
l_0: & & & cn & & & & cn \\
l_1: & & cn/10 & & & 9cn/10 & & cn \\
l_2: & cn/10^2 & & 9cn/10^2 & 9cn/10^2 & & 9^2cn/10^2 & cn \\
 & & & \cdots\cdots & & & & \\
l_h: & cn/10^h & & & & & c9^h n/10^h & cn \\
 & & & \cdots\cdots & & & & \\
l_k: & & & & & & c9^k n/10^k & \leq cn
\end{array}$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lccccccc}
l_0: & & & cn & & & & cn \\
l_1: & & cn/10 & & & 9cn/10 & & cn \\
l_2: & cn/10^2 & & 9cn/10^2 & 9cn/10^2 & & 9^2cn/10^2 & cn \\
 & & & \cdots\cdots & & & & \\
l_h: & cn/10^h & & & & & c9^h n/10^h & cn \\
 & & & \cdots\cdots & & & & \\
l_k: & & & & & & c9^k n/10^k & \leq cn
\end{array}$$

where $(\frac{1}{10})^h n = 1$, i.e., $h = \log_{10} n$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lccccccc}
l_0: & & & cn & & & & cn \\
l_1: & & cn/10 & & & 9cn/10 & & cn \\
l_2: & cn/10^2 & & 9cn/10^2 & 9cn/10^2 & & 9^2cn/10^2 & cn \\
 & & & \cdots\cdots & & & & \\
l_h: & cn/10^h & & & & & c9^h n/10^h & cn \\
 & & & \cdots\cdots & & & & \\
l_k: & & & & & & c9^k n/10^k & \leq cn
\end{array}$$

where $(\frac{1}{10})^h n = 1$, i.e., $h = \log_{10} n$

$(\frac{9}{10})^k n = 1$, i.e., $k = \log_{\frac{10}{9}} n$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lcccccc}
l_0: & & & cn & & & cn \\
l_1: & & cn/10 & & 9cn/10 & & cn \\
l_2: & cn/10^2 & 9cn/10^2 & 9cn/10^2 & 9^2cn/10^2 & & cn \\
 & & & \cdots\cdots & & & \\
l_h: & cn/10^h & & & & c9^hn/10^h & cn \\
 & & & \cdots\cdots & & & \\
l_k: & & & & & c9^kn/10^k & \leq cn
\end{array}$$

where $(\frac{1}{10})^h n = 1$, i.e., $h = \log_{10} n$

$(\frac{9}{10})^k n = 1$, i.e., $k = \log_{\frac{10}{9}} n$

$$cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lcccccc}
l_0: & & & cn & & & cn \\
l_1: & & cn/10 & & 9cn/10 & & cn \\
l_2: & cn/10^2 & 9cn/10^2 & 9cn/10^2 & 9^2cn/10^2 & & cn \\
 & & & \cdots\cdots & & & \\
l_h: & cn/10^h & & & & c9^hn/10^h & cn \\
 & & & \cdots\cdots & & & \\
l_k: & & & & & c9^kn/10^k & \leq cn
\end{array}$$

where $(\frac{1}{10})^h n = 1$, i.e., $h = \log_{10} n$

$(\frac{9}{10})^k n = 1$, i.e., $k = \log_{\frac{10}{9}} n$

$$cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n$$

$$T(n) \leq cn \log_{\frac{10}{9}} n$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lcccccc}
l_0: & & & cn & & & cn \\
l_1: & & cn/10 & & 9cn/10 & & cn \\
l_2: & cn/10^2 & 9cn/10^2 & 9cn/10^2 & 9^2cn/10^2 & & cn \\
 & & & \cdots\cdots & & & \\
l_h: & cn/10^h & & & & c9^hn/10^h & cn \\
 & & & \cdots\cdots & & & \\
l_k: & & & & & c9^kn/10^k & \leq cn
\end{array}$$

where $(\frac{1}{10})^h n = 1$, i.e., $h = \log_{10} n$

$(\frac{9}{10})^k n = 1$, i.e., $k = \log_{\frac{10}{9}} n$

$$cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n$$

$$T(n) \leq cn \log_{\frac{10}{9}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}}$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lccccccc}
l_0: & & & & cn & & & cn \\
l_1: & & cn/10 & & & 9cn/10 & & cn \\
l_2: & cn/10^2 & & 9cn/10^2 & & 9cn/10^2 & 9^2cn/10^2 & cn \\
 & & & & \cdots\cdots & & & \\
l_h: & cn/10^h & & & & & c9^h n/10^h & cn \\
 & & & & \cdots\cdots & & & \\
l_k: & & & & & & c9^k n/10^k & \leq cn
\end{array}$$

where $(\frac{1}{10})^h n = 1$, i.e., $h = \log_{10} n$

$(\frac{9}{10})^k n = 1$, i.e., $k = \log_{\frac{10}{9}} n$

$$cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n$$

$$T(n) \leq cn \log_{\frac{10}{9}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} = c' n \log_2 n$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lccccccc}
l_0: & & & & cn & & & cn \\
l_1: & & cn/10 & & & 9cn/10 & & cn \\
l_2: & cn/10^2 & & 9cn/10^2 & & 9cn/10^2 & 9^2cn/10^2 & cn \\
 & & & & \cdots\cdots & & & \\
l_h: & cn/10^h & & & & & c9^h n/10^h & cn \\
 & & & & \cdots\cdots & & & \\
l_k: & & & & & & c9^k n/10^k & \leq cn
\end{array}$$

where $(\frac{1}{10})^h n = 1$, i.e., $h = \log_{10} n$

$(\frac{9}{10})^k n = 1$, i.e., $k = \log_{\frac{10}{9}} n$

$$cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n$$

$$T(n) \leq cn \log_{\frac{10}{9}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} = c' n \log_2 n = O(n \log_2 n)$$

Chapter 7. Quicksort

What running time would it be if 10:90 partition is always guaranteed?

$$T(n) \leq T(n/10) + T(9n/10) + cn$$

Using the recursive-tree method, we have

$$\begin{array}{lcccccc}
l_0: & & & cn & & & cn \\
l_1: & & cn/10 & & 9cn/10 & & cn \\
l_2: & cn/10^2 & 9cn/10^2 & 9cn/10^2 & 9^2cn/10^2 & & cn \\
 & & & \cdots\cdots & & & \\
l_h: & cn/10^h & & & & c9^hn/10^h & cn \\
 & & & \cdots\cdots & & & \\
l_k: & & & & & c9^kn/10^k & \leq cn
\end{array}$$

where $(\frac{1}{10})^h n = 1$, i.e., $h = \log_{10} n$
$(\frac{9}{10})^k n = 1$, i.e., $k = \log_{\frac{10}{9}} n$

$$cn \log_{10} n \leq T(n) \leq cn \log_{\frac{10}{9}} n$$

$$T(n) \leq cn \log_{\frac{10}{9}} n = cn \frac{\log_2 n}{\log_2 \frac{10}{9}} = c'n \log_2 n = O(n \log_2 n)$$

where $c' = c/\log_2 \frac{10}{9}$

Chapter 7. Quicksort

Instead of analyzing QuickSort (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Chapter 7. Quicksort

Instead of analyzing QUICKSORT (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm RANDOMIZED-PARTITION(A, p, r)

1. $i = random(p, r)$
2. exchange $A[r] \longleftrightarrow A[i]$
3. **return** (PARTITON(A, p, r))

Chapter 7. Quicksort

Instead of analyzing QuickSort (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm Randomized-Partition(A, p, r)

1. $i = random(p, r)$
2. exchange $A[r] \longleftrightarrow A[i]$
3. **return** (Partiton(A, p, r))

Algorithm Randomized QuickSort (A, p, r)

Chapter 7. Quicksort

Instead of analyzing QUICKSORT (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm RANDOMIZED-PARTITION(A, p, r)
1. $i = random(p, r)$
2. exchange $A[r] \longleftrightarrow A[i]$
3. **return** (PARTITON(A, p, r))

Algorithm RANDOMIZED QUICKSORT (A, p, r)
1. **if** $p < r$

Chapter 7. Quicksort

Instead of analyzing QUICKSORT (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm RANDOMIZED-PARTITION(A, p, r)
1. $i = random(p, r)$
2. exchange $A[r] \longleftrightarrow A[i]$
3. **return** (PARTITON(A, p, r))

Algorithm RANDOMIZED QUICKSORT (A, p, r)
1. **if** $p < r$
2. **then** $q =$ RANDOMIZED-PARTITION(A, p, r)

Chapter 7. Quicksort

Instead of analyzing QUICKSORT (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm RANDOMIZED-PARTITION(A, p, r)
1. $i = random(p, r)$
2. exchange $A[r] \longleftrightarrow A[i]$
3. **return** (PARTITON(A, p, r))

Algorithm RANDOMIZED QUICKSORT (A, p, r)
1. **if** $p < r$
2. **then** $q =$ RANDOMIZED-PARTITION(A, p, r)
3. RANDOMIZED QUICKSORT $(A, p, q - 1)$

Chapter 7. Quicksort

Instead of analyzing QUICKSORT (with uniformly distributed for input) we design a randomized version of the algorithm and analyze it.

Algorithm RANDOMIZED-PARTITION(A, p, r)
1. $i = random(p, r)$
2. exchange $A[r] \longleftrightarrow A[i]$
3. **return** (PARTITON(A, p, r))

Algorithm RANDOMIZED QUICKSORT (A, p, r)
1. **if** $p < r$
2. **then** $q =$ RANDOMIZED-PARTITION(A, p, r)
3. RANDOMIZED QUICKSORT $(A, p, q-1)$
4. RANDOMIZED QUICKSORT $(A, q+1, r)$

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Observation 1: x_i is compared with x_j only when either is a pivot;

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Observation 1: x_i is compared with x_j only when either is a pivot;

Observation 2: x_i is compared with x_j at most once;

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Observation 1: x_i is compared with x_j only when either is a pivot;

Observation 2: x_i is compared with x_j at most once;

- define random variable $X_{i,j}$ indicating
  if a comparison between x_i and x_j occurs.

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Observation 1: x_i is compared with x_j only when either is a pivot;

Observation 2: x_i is compared with x_j at most once;

- define random variable $X_{i,j}$ indicating
  if a comparison between x_i and x_j occurs.
- $X_{i,j} \in \{0, 1\}$

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Observation 1: x_i is compared with x_j only when either is a pivot;

Observation 2: x_i is compared with x_j at most once;

- define random variable $X_{i,j}$ indicating
  if a comparison between x_i and x_j occurs.
- $X_{i,j} \in \{0, 1\}$
- let $X = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} X_{i,j}$, written as $X = \sum_{i<j} X_{i,j}$

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Observation 1: x_i is compared with x_j only when either is a pivot;

Observation 2: x_i is compared with x_j at most once;

- define random variable $X_{i,j}$ indicating
  if a comparison between x_i and x_j occurs.
- $X_{i,j} \in \{0, 1\}$
- let $X = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} X_{i,j}$, written as $X = \sum_{i<j} X_{i,j}$
- the expected number of comparisons is

$$E(X) = E(\sum_{i<j} X_{i,j})$$

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Observation 1: x_i is compared with x_j only when either is a pivot;

Observation 2: x_i is compared with x_j at most once;

- define random variable $X_{i,j}$ indicating
  if a comparison between x_i and x_j occurs.
- $X_{i,j} \in \{0, 1\}$
- let $X = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} X_{i,j}$, written as $X = \sum_{i<j} X_{i,j}$
- the expected number of comparisons is

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) =$$

Chapter 7. Quicksort

Analysis of RANDOMIZED-QUICKSORT

- count the expected number of comparisons between x_i and x_j;

Observation 1: x_i is compared with x_j only when either is a pivot;

Observation 2: x_i is compared with x_j at most once;

- define random variable $X_{i,j}$ indicating
  if a comparison between x_i and x_j occurs.
- $X_{i,j} \in \{0, 1\}$
- let $X = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} X_{i,j}$, written as $X = \sum_{i<j} X_{i,j}$
- the expected number of comparisons is

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} P(X_{i,j} = 1)$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j})$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) =$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

(1) x_i, x_j are in the same sublist L;

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

(1) x_i, x_j are in the same sublist L;
(2) either is chosen to be the pivot;

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

(1) x_i, x_j are in the same sublist L;
(2) either is chosen to be the pivot;

$P(X_{i,j} = 1) = 2\frac{1}{|L|}$, where $|L|$ is the size of the sublist. why?

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

(1) x_i, x_j are in the same sublist L;
(2) either is chosen to be the pivot;

$P(X_{i,j} = 1) = 2\frac{1}{|L|}$, where $|L|$ is the size of the sublist. why?

but we do not know the size of the sublist L!

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

(1) x_i, x_j are in the same sublist L;
(2) either is chosen to be the pivot;

$P(X_{i,j} = 1) = 2\frac{1}{|L|}$, where $|L|$ is the size of the sublist. why?

but we do not know the size of the sublist L!

however, if x_i, x_j are so indexed in the final sorted list,

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

(1) x_i, x_j are in the same sublist L;
(2) either is chosen to be the pivot;

$P(X_{i,j} = 1) = 2\frac{1}{|L|}$, where $|L|$ is the size of the sublist. why?

but we do not know the size of the sublist L!

however, if x_i, x_j are so indexed in the final sorted list, then

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

(1) x_i, x_j are in the same sublist L;
(2) either is chosen to be the pivot;

$P(X_{i,j} = 1) = 2\frac{1}{|L|}$, where $|L|$ is the size of the sublist. why?

but we do not know the size of the sublist L!

however, if x_i, x_j are so indexed in the final sorted list, then

size of the sublist (which x_i, x_j belongs to)
$|L| \geq (j - i + 1)$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$X_{i,j} = 1$, i.e., comparison between x_i and x_j occurs only when

(1) x_i, x_j are in the same sublist L;
(2) either is chosen to be the pivot;

$P(X_{i,j} = 1) = 2\frac{1}{|L|}$, where $|L|$ is the size of the sublist. why?

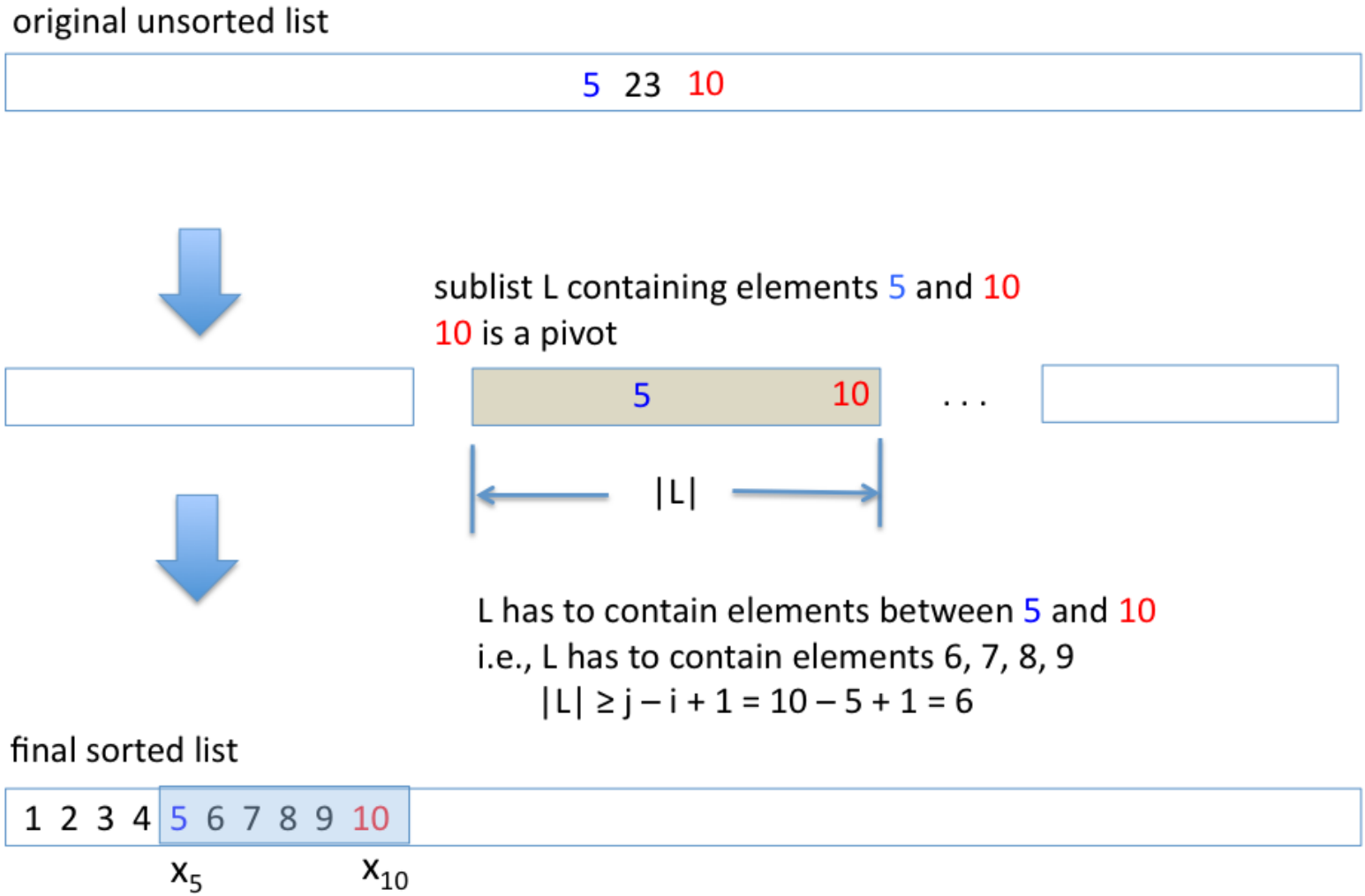
but we do not know the size of the sublist L!

however, if x_i, x_j are so indexed in the final sorted list, then

size of the sublist (which x_i, x_j belongs to)
$|L| \geq (j - i + 1)$

So $P(X_{i,j} = 1) \leq 2\frac{1}{|L|} \leq 2\frac{1}{j-i+1}$

Chapter 7. Quicksort



Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$$\leq \sum_{i<j} 2\frac{1}{j-i+1}$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$$\leq \sum_{i<j} 2\frac{1}{j-i+1} = \sum_{i=1}^{n-1} \sum_{j=2}^{n} 2\frac{1}{j-i+1}$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$$\leq \sum_{i<j} 2\frac{1}{j-i+1} = \sum_{i=1}^{n-1}\sum_{j=2}^{n} 2\frac{1}{j-i+1}$$

$$\leq \sum_{i=1}^{n-1} 2\sum_{k=1}^{n-1} \frac{1}{k+1} \leq$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$$\leq \sum_{i<j} 2\frac{1}{j-i+1} = \sum_{i=1}^{n-1}\sum_{j=2}^{n} 2\frac{1}{j-i+1}$$

$$\leq \sum_{i=1}^{n-1} 2\sum_{k=1}^{n-1} \frac{1}{k+1} \leq \sum_{i=1}^{n-1} c\log_2 n$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$$\leq \sum_{i<j} 2\frac{1}{j-i+1} = \sum_{i=1}^{n-1}\sum_{j=2}^{n} 2\frac{1}{j-i+1}$$

$$\leq \sum_{i=1}^{n-1} 2\sum_{k=1}^{n-1} \frac{1}{k+1} \leq \sum_{i=1}^{n-1} c\log_2 n \leq cn\log_2 n$$

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

$$\leq \sum_{i<j} 2\frac{1}{j-i+1} = \sum_{i=1}^{n-1}\sum_{j=2}^{n} 2\frac{1}{j-i+1}$$

$$\leq \sum_{i=1}^{n-1} 2\sum_{k=1}^{n-1} \frac{1}{k+1} \leq \sum_{i=1}^{n-1} c\log_2 n \leq cn\log_2 n$$

for some constant $c > 0$.

Analysis of RANDOMIZED-QUICKSORT (cont.)

$$E(X) = E(\sum_{i<j} X_{i,j}) = \sum_{i<j} E(X_{i,j}) = \sum_{i<j} Prob(X_{i,j} = 1)$$

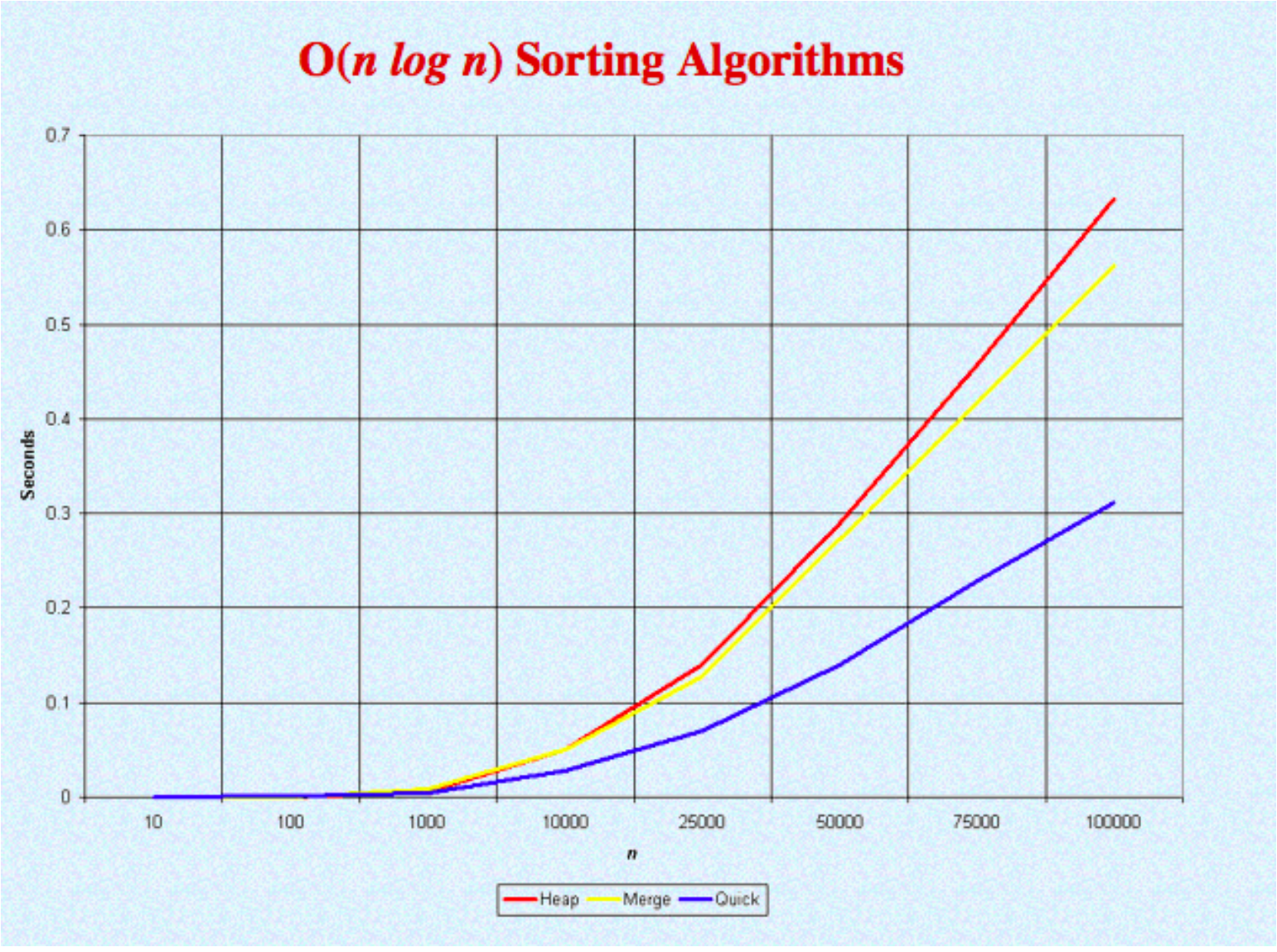
$$\leq \sum_{i<j} 2\frac{1}{j-i+1} = \sum_{i=1}^{n-1}\sum_{j=2}^{n} 2\frac{1}{j-i+1}$$

$$\leq \sum_{i=1}^{n-1} 2\sum_{k=1}^{n-1} \frac{1}{k+1} \leq \sum_{i=1}^{n-1} c\log_2 n \leq cn\log_2 n$$

for some constant $c > 0$.

So $E(X) = O(n\log_2 n)$.

Chapter 7. Quicksort



Chapter 8. Lower bounds and sorting in linear time

Chapter 8. Lower Bounds and Sorting in Linear Time

Chapter 8. Lower bounds and sorting in linear time

- We have used Big-O for upper bounds.

Chapter 8. Lower Bounds and Sorting in Linear Time

Chapter 8. Lower bounds and sorting in linear time

- We have used Big-O for upper bounds.
- We need another notation for lower bounds.

Chapter 8. Lower Bounds and Sorting in Linear Time

Chapter 8. Lower bounds and sorting in linear time

- We have used Big-O for upper bounds.
- We need another notation for lower bounds.

Define $\Omega(g(n)$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

Chapter 8. Lower bounds and sorting in linear time

- We have used Big-O for upper bounds.
- We need another notation for lower bounds.

Define $\Omega(g(n)$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$$\Omega(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k\}$$

In other word, $\lim_{n\to\infty} \frac{f(n)}{g(n)} =$

Chapter 8. Lower bounds and sorting in linear time

- We have used Big-O for upper bounds.
- We need another notation for lower bounds.

Define $\Omega(g(n)$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$$\Omega(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k\}$$

$$\text{In other word, } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{ constant } > 0 \text{ or}$$

Chapter 8. Lower bounds and sorting in linear time

- We have used Big-O for upper bounds.
- We need another notation for lower bounds.

Define $\Omega(g(n)$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$$\Omega(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k\}$$

$$\text{In other word, } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{ constant } > 0 \text{ or } \infty$$

Chapter 8. Lower bounds and sorting in linear time

- We have used Big-O for upper bounds.
- We need another notation for lower bounds.

Define $\Omega(g(n)$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$$\Omega(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k\}$$

$$\text{In other word, } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{ constant } > 0 \text{ or } \infty$$

For example, we have shown $T(n) = \Omega(n^2)$ for Insertion Sort.

Chapter 8. Lower bounds and sorting in linear time

- We have used Big-O for upper bounds.
- We need another notation for lower bounds.

Define $\Omega(g(n)$ be the set of functions that have growth rates not slower than $cg(n)$ for any given constant $c > 0$.

$$\Omega(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq k\}$$

$$\text{In other word, } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{ constant } > 0 \text{ or } \infty$$

For example, we have shown $T(n) = \Omega(n^2)$ for Insertion Sort.

Proof techniques for Big-Ω are similar to those for Big-O.

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms	Sorting Problem

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms	Sorting Problem
upper bound	Insertion Sort $O(n^2)$	

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$		

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$
lower bound	INSERTION SORT $\Omega(n^2)$		

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$
lower bound	INSERTION SORT $\Omega(n^2)$	$\longleftarrow$	$\Omega(n \log_2 n)$

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$
lower bound	INSERTION SORT $\Omega(n^2)$	$\longleftarrow$	$\Omega(n \log_2 n)$
	MERGE SORT $\Omega(n \log_2 n)$		

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$
lower bound	INSERTION SORT $\Omega(n^2)$	$\longleftarrow$	$\Omega(n \log_2 n)$
	MERGE SORT $\Omega(n \log_2 n)$	$\longleftarrow$	$\Omega(n \log_2 n)$

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$
lower bound	INSERTION SORT $\Omega(n^2)$	$\longleftarrow$	$\Omega(n \log_2 n)$
	MERGE SORT $\Omega(n \log_2 n)$	$\longleftarrow$	$\Omega(n \log_2 n)$

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ if and only if $f(n) = \Theta(g(n))$.

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$
lower bound	INSERTION SORT $\Omega(n^2)$	$\longleftarrow$	$\Omega(n \log_2 n)$
	MERGE SORT $\Omega(n \log_2 n)$	$\longleftarrow$	$\Omega(n \log_2 n)$

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ if and only if $f(n) = \Theta(g(n))$.

So we say

- INSERTION SORT runs in time $\Theta(n^2)$,

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$
lower bound	INSERTION SORT $\Omega(n^2)$	$\longleftarrow$	$\Omega(n \log_2 n)$
	MERGE SORT $\Omega(n \log_2 n)$	$\longleftarrow$	$\Omega(n \log_2 n)$

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ if and only if $f(n) = \Theta(g(n))$.

So we say

- INSERTION SORT runs in time $\Theta(n^2)$,
- MERGESORT runs in time $\Theta(n \log_2 n)$,

Chapter 8. Lower Bounds and Sorting in Linear Time

Important notes on lower bound and upper bound

Bounds	Algorithms		Sorting Problem
upper bound	INSERTION SORT $O(n^2)$	$\longrightarrow$	$O(n^2)$
	MERGE SORT $O(n \log_2 n)$	$\longrightarrow$	$O(n \log_2 n)$
lower bound	INSERTION SORT $\Omega(n^2)$	$\longleftarrow$	$\Omega(n \log_2 n)$
	MERGE SORT $\Omega(n \log_2 n)$	$\longleftarrow$	$\Omega(n \log_2 n)$

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ if and only if $f(n) = \Theta(g(n))$.

So we say

- INSERTION SORT runs in time $\Theta(n^2)$,
- MERGESORT runs in time $\Theta(n \log_2 n)$,
- the sorting problem has $\Theta(n \log_2 n)$ time complexity.

Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

- each internal node denotes $(x_i \leq x_j)$, with two outcomes

Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

- each internal node denotes $(x_i \leq x_j)$, with two outcomes
- each path corresponds to one possible outcome of the algorithm

Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

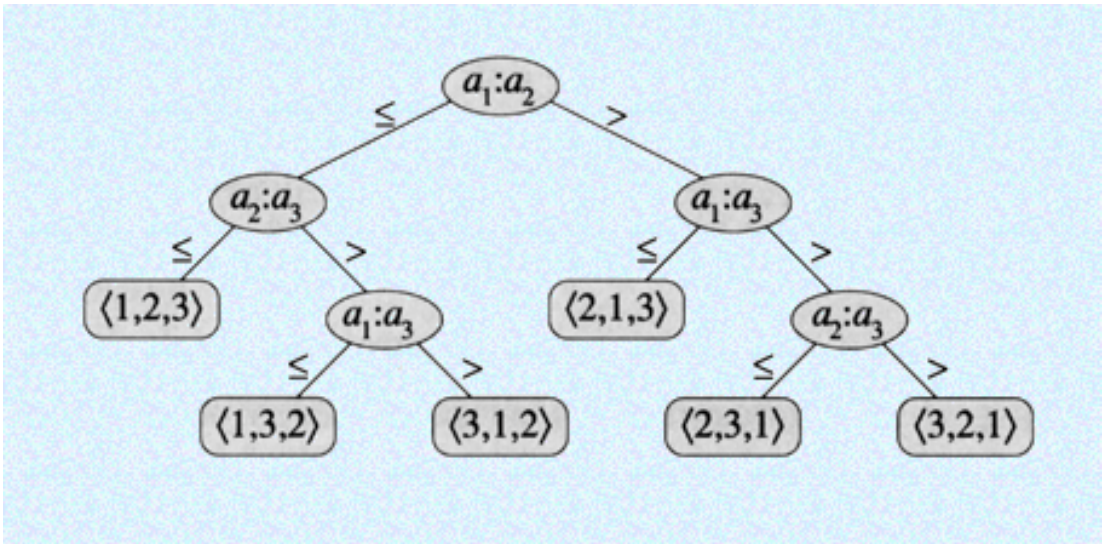
- each internal node denotes $(x_i \leq x_j)$, with two outcomes
- each path corresponds to one possible outcome of the algorithm
- each path is for one permutation of generic list $(1, 2, \ldots, n)$

Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

- each internal node denotes $(x_i \leq x_j)$, with two outcomes
- each path corresponds to one possible outcome of the algorithm
- each path is for one permutation of generic list $(1, 2, \ldots, n)$

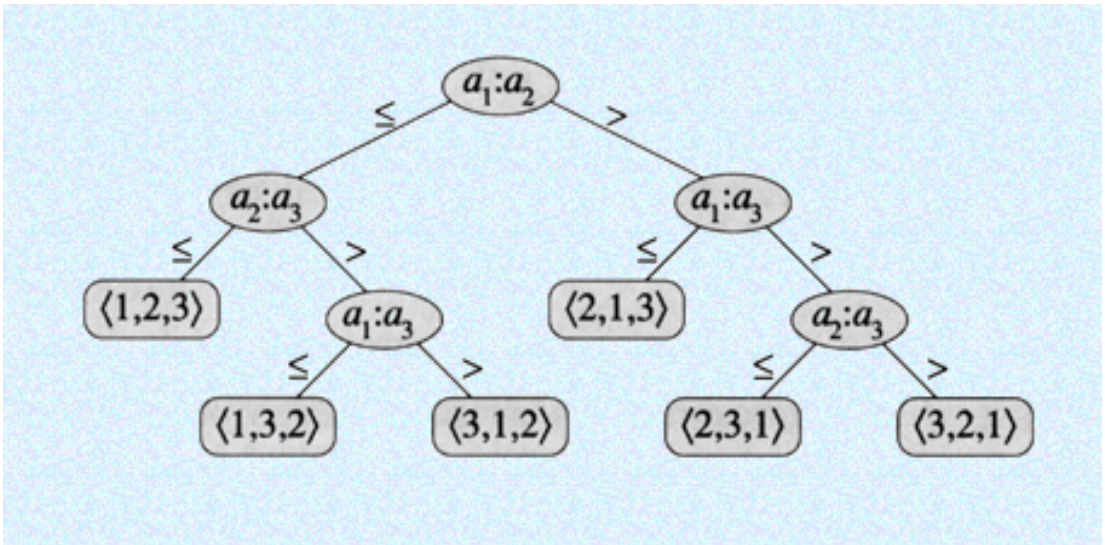


Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

- each internal node denotes $(x_i \leq x_j)$, with two outcomes
- each path corresponds to one possible outcome of the algorithm
- each path is for one permutation of generic list $(1, 2, \ldots, n)$



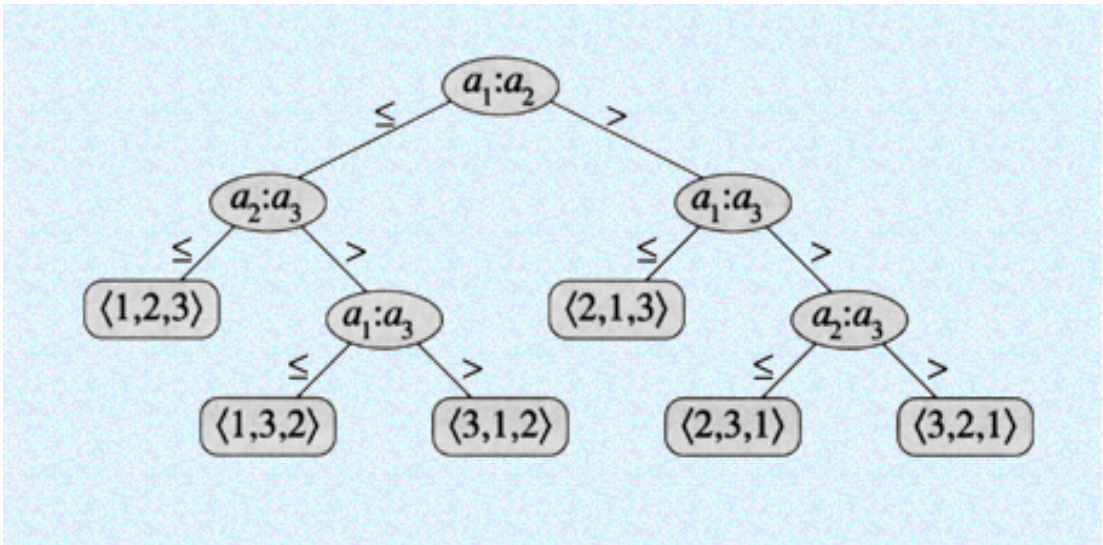
Claim 1: total number of leaves is $\geq n!$.

Chapter 8. Lower Bounds and Sorting in Linear Time

Deriving a lower bound for sorting

with decision tree as algorithm/computation model

- each internal node denotes $(x_i \leq x_j)$, with two outcomes
- each path corresponds to one possible outcome of the algorithm
- each path is for one permutation of generic list $(1, 2, \ldots, n)$



Claim 1: total number of leaves is $\geq n!$.

Claim 2: the height of the tree at least $\geq \log n!$.
(The minimum of heights of all such trees!)

Chapter 8. Lower Bounds and Sorting in Linear Time

Theorem: Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.

Chapter 8. Lower Bounds and Sorting in Linear Time

Theorem: Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.

Prove.
The longest path from the root to a leave is $\Omega(\log n!)$. I.e.,
the number of comparisons needed in the worst case is $\Omega(\log n!)$.

Chapter 8. Lower Bounds and Sorting in Linear Time

Theorem: Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.

Prove.
The longest path from the root to a leave is $\Omega(\log n!)$. I.e.,
the number of comparisons needed in the worst case is $\Omega(\log n!)$.

$$n! = n(n-1)(n-2)\cdots(n-\frac{n}{2})(n-\frac{n}{2}-1)\cdots 2\times 1$$

$$\geq (\frac{n}{2})^{\frac{n}{2}} \times 2^{\frac{n}{2}-1} \geq \frac{1}{2}n^{\frac{n}{2}}$$

or by Stirling's formula:

$$n! = \sqrt{2\pi n}(n/e)^n(1+O(1/n))$$

$$\Omega(\log(n!)) = \Omega(n \log n)$$

Chapter 8. Lower Bounds and Sorting in Linear Time

Sorting algorithms with worst case linear time

Chapter 8. Lower Bounds and Sorting in Linear Time

Sorting algorithms with worst case linear time

- count sort
- radix sort
- bucket sort

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$
8. $\{C[i]$ contains the number of elements whose values $\leq i\}$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$
8. $\{C[i]$ contains the number of elements whose values $\leq i\}$
9. **for** $j = length[A]$ **downto** 1

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$
8. $\{C[i]$ contains the number of elements whose values $\leq i\}$
9. **for** $j = length[A]$ **downto** 1
10. $\quad B[C[A[j]]] = A[j]$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$
8. $\{C[i]$ contains the number of elements whose values $\leq i\}$
9. **for** $j = length[A]$ **downto** 1
10. $\quad B[C[A[j]]] = A[j]$
11. $\quad C[A[j]] = C[A[j]] - 1$

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$
8. $\{C[i]$ contains the number of elements whose values $\leq i\}$
9. **for** $j = length[A]$ **downto** 1
10. $\quad B[C[A[j]]] = A[j]$
11. $\quad C[A[j]] = C[A[j]] - 1$

Example: A: 2 5 3 0 2 3 0 3, $\quad k = 5$,

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$
8. $\{C[i]$ contains the number of elements whose values $\leq i\}$
9. **for** $j = length[A]$ **downto** 1
10. $\quad B[C[A[j]]] = A[j]$
11. $\quad C[A[j]] = C[A[j]] - 1$

Example: A: 2 5 3 0 2 3 0 3, $\quad k = 5,$ $\quad$ C: 2 0 2 3 0 1

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$
8. $\{C[i]$ contains the number of elements whose values $\leq i\}$
9. **for** $j = length[A]$ **downto** 1
10. $\quad B[C[A[j]]] = A[j]$
11. $\quad C[A[j]] = C[A[j]] - 1$

Example: A: 2 5 3 0 2 3 0 3, $k = 5$, C: 2 0 2 3 0 1

analysis:

Chapter 8. Lower Bounds and Sorting in Linear Time

Count sort

Algorithm COUNTING-SORT (A, B, k) $\{A$ contains n integers; k is the max$\}$

1. **for** $i = 0$ **to** k
2. $\quad C[i] = 0$
3. **for** $j = 1$ **to** $length[A]$
4. $\quad C[A[j]] = C[A[j]] + 1$
5. $\{C[i]$ contains the number of elements whose values $= i\}$
6. **for** $i = 1$ **to** k
7. $\quad C[i] = C[i] + C[i-1]$
8. $\{C[i]$ contains the number of elements whose values $\leq i\}$
9. **for** $j = length[A]$ **downto** 1
10. $\quad B[C[A[j]]] = A[j]$
11. $\quad C[A[j]] = C[A[j]] - 1$

Example: A: 2 5 3 0 2 3 0 3, $k = 5$, C: 2 0 2 3 0 1

analysis: $T(n) = O(k + n)$

Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm RADIX-SORT(A, d)

Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm RADIX-SORT(A, d)

1. **for** $i = 1$ **to** d

Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm RADIX-SORT(A, d)

1. **for** $i = 1$ **to** d
2. **sort** A on the ith digit

Chapter 8. Lower Bounds and Sorting in Linear Time

Radix Sort:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm RADIX-SORT(A, d)

1. **for** $i = 1$ **to** d
2. **sort** A on the ith digit

Lemma. Given n b-bit binary numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Chapter 8. Lower Bounds and Sorting in Linear Time

Lemma. Given n b-bit binary numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Chapter 8. Lower Bounds and Sorting in Linear Time

Lemma. Given n b-bit binary numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Proof. Each b-digit binary number can be regarded as $\lceil b/r \rceil$ r-digit binary numbers. These r-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Chapter 8. Lower Bounds and Sorting in Linear Time

Lemma. Given n b-bit binary numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Proof. Each b-digit binary number can be regarded as $\lceil b/r \rceil$ r-digit binary numbers. These r-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run RADIX-SORT on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.

Chapter 8. Lower Bounds and Sorting in Linear Time

Lemma. Given n b-bit binary numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Proof. Each b-digit binary number can be regarded as $\lceil b/r \rceil$ r-digit binary numbers. These r-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run RADIX-SORT on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.

For every column, sorting by COUNTING-SORT with $2^r - 1$ being the maximum.

Chapter 8. Lower Bounds and Sorting in Linear Time

Lemma. Given n b-bit binary numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Proof. Each b-digit binary number can be regarded as $\lceil b/r \rceil$ r-digit binary numbers. These r-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run RADIX-SORT on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.

For every column, sorting by COUNTING-SORT with $2^r - 1$ being the maximum.

The total time is $O(\lceil b/r \rceil (n + 2^r))$, where $(n + 2^r)$ is time for COUNTING-SORT.

Chapter 8. Lower Bounds and Sorting in Linear Time

Lemma. Given n b-bit binary numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Proof. Each b-digit binary number can be regarded as $\lceil b/r \rceil$ r-digit binary numbers. These r-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run RADIX-SORT on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.

For every column, sorting by COUNTING-SORT with $2^r - 1$ being the maximum.

The total time is $O(\lceil b/r \rceil (n + 2^r))$, where $(n + 2^r)$ is time for COUNTING-SORT.

Since all steps in the two algorithms are mandatory, the total time is also $\Omega(\lceil b/r \rceil (n + 2^r))$, thus $\Theta(\lceil b/r \rceil (n + 2^r))$.

Chapter 8. Lower Bounds and Sorting in Linear Time

Lemma. Given n b-bit binary numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta(\lceil b/r \rceil (n + 2^r))$ time.

Proof. Each b-digit binary number can be regarded as $\lceil b/r \rceil$ r-digit binary numbers. These r-digit binary numbers are of integer values in the range of $\{0, 1, \ldots, 2^r - 1\}$.

Run RADIX-SORT on the original binary numbers assumed to be $\lceil b/r \rceil$ columns.

For every column, sorting by COUNTING-SORT with $2^r - 1$ being the maximum.

The total time is $O(\lceil b/r \rceil (n + 2^r))$, where $(n + 2^r)$ is time for COUNTING-SORT.

Since all steps in the two algorithms are mandatory, the total time is also $\Omega(\lceil b/r \rceil (n + 2^r))$, thus $\Theta(\lceil b/r \rceil (n + 2^r))$.

Once b and n are given, we can choose r to minimize the quantity $\lceil b/r \rceil (n + 2^r)$.

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n - 1]$

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n - 1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 $\rightarrow$ .12 $\rightarrow$ .17

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 $\rightarrow$ .12 $\rightarrow$ .17
2 $\rightarrow$ .21 $\rightarrow$ .23 $\rightarrow$ .26

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)

1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /
5 /

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)

1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /
5 /
6 → .68

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /
5 /
6 → .68
7 → .72 → .78

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /
5 /
6 → .68
7 → .72 → .78
8 /

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /
5 /
6 → .68
7 → .72 → .78
8 /
9 → .94

Chapter 8. Lower Bounds and Sorting in Linear Time

Bucket Sort (assuming uniform distribution of inputs)

Algorithm BUCKET-SORT(A)
1. $n = length[A]$
2. **for** $i = 1$ **to** n
3. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i = 0$ **to** $n - 1$
5. sort list $B[i]$ with INSERTION SORT
6. concatenate the lists $B[0], B[1], ..., B[n-1]$

A: .78 .17 .39 .26 .72 .94 .21 .12 .23 .68

B: 0 /
1 → .12 → .17
2 → .21 → .23 → .26
3 → .39
4 /
5 /
6 → .68
7 → .72 → .78
8 /
9 → .94

Chapter 9. Medians and order statistics

Chapter 9. Medians and order statistics

- find the maximum: linear time

Chapter 9. Medians and order statistics

- find the maximum: linear time
- find the minimum: linear time

Chapter 9. Medians and Order Statistics

Chapter 9. Medians and order statistics

- find the maximum: linear time
- find the minimum: linear time
- find the median (i.e., the $\frac{n}{2}$th smallest element) ?

Chapter 9. Medians and Order Statistics

Chapter 9. Medians and order statistics

- find the maximum: linear time
- find the minimum: linear time
- find the median (i.e., the $\frac{n}{2}$th smallest element) ?

  the problem has upper bound $O(n \log_2 n)$.

Chapter 9. Medians and Order Statistics

Chapter 9. Medians and order statistics

- find the maximum: linear time
- find the minimum: linear time
- find the median (i.e., the $\frac{n}{2}$th smallest element) ?

  the problem has upper bound $O(n \log_2 n)$. why?

Chapter 9. Medians and Order Statistics

Chapter 9. Medians and order statistics

- find the maximum: linear time
- find the minimum: linear time
- find the median (i.e., the $\frac{n}{2}$th smallest element) ?

  the problem has upper bound $O(n \log_2 n)$. why?

  Can we do better?

Chapter 9. Medians and Order Statistics

Chapter 9. Medians and Order Statistics

Selection problem

Chapter 9. Medians and Order Statistics

Selection problem

INPUT: a list A of elements, an integer i;

Chapter 9. Medians and Order Statistics

Selection problem

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

Chapter 9. Medians and Order Statistics

Selection problem

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

There are algorithms solving it in linear time.

Chapter 9. Medians and Order Statistics

Selection problem

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

There are algorithms solving it in linear time.

Two types of algorithms:

Chapter 9. Medians and Order Statistics

Selection problem

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

There are algorithms solving it in linear time.

Two types of algorithms:

- Selection in *expected* linear time (but worst case $\Theta(n^2)$)

Chapter 9. Medians and Order Statistics

Selection problem

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

There are algorithms solving it in linear time.

Two types of algorithms:

- Selection in *expected* linear time (but worst case $\Theta(n^2)$)
- Selection in worst case linear time

Chapter 9. Medians and Order Statistics

Selection in *expected* linear time

Chapter 9. Medians and Order Statistics

Selection in *expected* linear time

INPUT: a list A of elements, an integer i;

Chapter 9. Medians and Order Statistics

Selection in *expected* linear time

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

Chapter 9. Medians and Order Statistics

Selection in *expected* linear time

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

Idea of the algorithm:

- randomly identify a pivot x and partition the list A into two sublists A_l and A_u; assume the rank of x is k;

Chapter 9. Medians and Order Statistics

Selection in *expected* linear time

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

Idea of the algorithm:

- randomly identify a pivot x and partition the list A into two sublists A_l and A_u; assume the rank of x is k;
- if $i = k$, done, return (x);

Chapter 9. Medians and Order Statistics

Selection in *expected* linear time

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

Idea of the algorithm:

- randomly identify a pivot x and partition the list A into two sublists A_l and A_u; assume the rank of x is k;
- if $i = k$, done, return (x);

  else if $k > i$, recursively do for A_l with i;

Chapter 9. Medians and Order Statistics

Selection in *expected* linear time

INPUT: a list A of elements, an integer i;
OUTPUT: the ith smallest element in A;

Idea of the algorithm:

- randomly identify a pivot x and partition the list A into two sublists A_l and A_u; assume the rank of x is k;
- if $i = k$, done, return (x);

  else if $k > i$, recursively do for A_l with i;

    else recursively do for A_u with $i - k$;

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$
5. **if** $i = k$

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$
5. **if** $i = k$
6. **return** $(A[q])$

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$
5. **if** $i = k$
6. **return** $(A[q])$
7. **else if** $i < k$

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$
5. **if** $i = k$
6. **return** $(A[q])$
7. **else if** $i < k$
8. **return** (RANDOMIZED-SELECT $(A, p, q - 1, i)$)

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$
5. **if** $i = k$
6. **return** $(A[q])$
7. **else if** $i < k$
8. **return** (RANDOMIZED-SELECT $(A, p, q - 1, i)$)
9. **else return** (RANDOMIZED-SELECT $(A, q + 1, r, i - k)$)

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$
5. **if** $i = k$
6. **return** $(A[q])$
7. **else if** $i < k$
8. **return** (RANDOMIZED-SELECT $(A, p, q - 1, i)$)
9. **else return** (RANDOMIZED-SELECT $(A, q + 1, r, i - k)$)

If pivots always partition lists into $\frac{n}{r} : \frac{r-1}{r}n$, for some $r > 1$,

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$
5. **if** $i = k$
6. **return** $(A[q])$
7. **else if** $i < k$
8. **return** (RANDOMIZED-SELECT $(A, p, q - 1, i)$)
9. **else return** (RANDOMIZED-SELECT $(A, q + 1, r, i - k)$)

If pivots always partition lists into $\frac{n}{r} : \frac{r-1}{r}n$, for some $r > 1$,

time $T(n)$ would have the recurrence

$$T(n) \leq \max\{T(\frac{n}{r}), T(\frac{(r-1)n}{r})\} + nc$$

Chapter 9. Medians and Order Statistics

Algorithm RANDOMIZED-SELECT (A, p, r, i)

1. **if** $p = r$
2. **return** $(A[p])$
3. $q =$ RANDOMIZED PARTITION (A, p, r)
4. $k = q - p + 1$
5. **if** $i = k$
6. **return** $(A[q])$
7. **else if** $i < k$
8. **return** (RANDOMIZED-SELECT $(A, p, q - 1, i)$)
9. **else return** (RANDOMIZED-SELECT $(A, q + 1, r, i - k)$)

If pivots always partition lists into $\frac{n}{r} : \frac{r-1}{r}n$, for some $r > 1$,

time $T(n)$ would have the recurrence

$$T(n) \leq \max\{T(\frac{n}{r}), T(\frac{(r-1)n}{r})\} + nc \leq T(\frac{(r-1)n}{r}) + cn$$

assuming $r \geq 2$,

$$T(n) \leq cn\frac{(r-1)}{r} + cn(\frac{(r-1)}{r})^2 + cn(\frac{(r-1)}{r})^4 + \ldots cn(\frac{(r-1)}{r})^m = O(n)$$

where $(\frac{(r-1)}{r})^m n = 1$, $m = \log_{\frac{r}{r-1}} n$

Chapter 9. Medians and Order Statistics

Performance analysis

Chapter 9. Medians and Order Statistics

Performance analysis

The worst case: running time $\Theta(n^2)$.

Chapter 9. Medians and Order Statistics

Performance analysis

The worst case: running time $\Theta(n^2)$.

Average case: $E[T(n)]$

Chapter 9. Medians and Order Statistics

Performance analysis

The worst case: running time $\Theta(n^2)$.

Average case: $E[T(n)]$

- on sublist $A[p..r]$, assume $n = r - p + 1$;

Chapter 9. Medians and Order Statistics

Performance analysis

The worst case: running time $\Theta(n^2)$.

Average case: $E[T(n)]$

- on sublist $A[p..r]$, assume $n = r - p + 1$;
- the algorithm identifies a pivot and recursively computes on sublist $A[p..q]$ (or $A[q + 1..r]$);

Chapter 9. Medians and Order Statistics

Performance analysis

The worst case: running time $\Theta(n^2)$.

Average case: $E[T(n)]$

- on sublist $A[p..r]$, assume $n = r - p + 1$;
- the algorithm identifies a pivot and recursively computes on sublist $A[p..q]$ (or $A[q+1..r]$);
- the pivot is chosen with probability $\frac{1}{n}$;

Average case: $E[T(n)]$ (cont')

- so the expected time $E[T(n)]$ needs to include the average time of recursion on the case when sublist $A[p..q]$ possibly has lengths $k = 0, 1, 2, \ldots, n-1$

Chapter 9. Medians and Order Statistics

Average case: $E[T(n)]$ (cont')

- so the expected time $E[T(n)]$ needs to include the average time of recursion on the case when sublist $A[p..q]$ possibly has lengths $k = 0, 1, 2, \ldots, n-1$
- thus the expected time $E[T(n)]$ is computed as

$$E[T(n)] \leq \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max\{k-1, n-k\})] + an, \text{ for some constant } a > 0$$

Chapter 9. Medians and Order Statistics

Average case: $E[T(n)]$ (cont')

- so the expected time $E[T(n)]$ needs to include the average time of recursion on the case when sublist $A[p..q]$ possibly has lengths $k = 0, 1, 2, \ldots, n-1$
- thus the expected time $E[T(n)]$ is computed as

$$E[T(n)] \leq \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max\{k-1, n-k\})] + an, \text{ for some constant } a > 0$$

because $\max\{k-1, n-k\} = k-1$ if $k > n/2$ and
$\max\{k-1, n-k\} = n-k$ if $k \leq n/2$

$$E[T(n)] \leq 2/n \Sigma_{k=n/2}^{n-1} E[T(k)] + an$$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method).

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n =$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n - 1$, $E[T(k)] \leq ck$;

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n - 1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n-1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\Sigma_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n}[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k] + an$$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n-1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\Sigma_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n}[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k] + an = \frac{2c}{n}[\frac{n-1}{2}(n) - \frac{n/2-1}{2}(n/2)] + an$$

$$= \cdots =$$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n-1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\Sigma_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n}[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k] + an = \frac{2c}{n}[\frac{n-1}{2}(n) - \frac{n/2-1}{2}(n/2)] + an$$

$$= \cdots = 3cn/4 + c/2 + an$$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n-1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\Sigma_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n}[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k] + an = \frac{2c}{n}[\frac{n-1}{2}(n) - \frac{n/2-1}{2}(n/2)] + an$$

$$= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n-1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\Sigma_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n}[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k] + an = \frac{2c}{n}[\frac{n-1}{2}(n) - \frac{n/2-1}{2}(n/2)] + an$$

$$= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$

when $(cn/4 - c/2 - an) \geq 0$.

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n-1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\Sigma_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n}[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k] + an = \frac{2c}{n}[\frac{n-1}{2}(n) - \frac{n/2-1}{2}(n/2)] + an$$

$$= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$

when $(cn/4 - c/2 - an) \geq 0$. That is $n \geq 2c/(c-4a)$

Chapter 9. Medians and Order Statistics

We conclude that $E[T(n)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an$

Theorem. $E[T(n)] = O(n)$.

Proof (by substitution method). We will prove that $E[T(n)] \leq cn$ for some $c > 0$.

- Base case: $n = ?$, we will decide later;
- Assumption: for all $k \leq n-1$, $E[T(k)] \leq ck$;
- Induction:

$$E[T(k)] \leq 2/n\Sigma_{k=n/2}^{n-1} E[T(k)] + an \leq 2/n\Sigma_{k=n/2}^{n-1} ck + an$$

$$= \frac{2c}{n}[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k] + an = \frac{2c}{n}[\frac{n-1}{2}(n) - \frac{n/2-1}{2}(n/2)] + an$$

$$= \cdots = 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an) \leq cn$$

when $(cn/4 - c/2 - an) \geq 0$. That is $n \geq 2c/(c-4a)$

- Base case: assume $T(n) \leq cn$, for $n < 2c/(c-4a)$

Chapter 9. Medians and Order Statistics

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

INPUT: set S of n elements and i;
OUTPUT: the ith smallest element in S;

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

INPUT: set S of n elements and i;
OUTPUT: the ith smallest element in S;

Main idea:

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

INPUT: set S of n elements and i;
OUTPUT: the ith smallest element in S;

Main idea:

- find a pivot x to partition the list S into two sublists S_1 and S_2,

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

INPUT: set S of n elements and i;
OUTPUT: the ith smallest element in S;

Main idea:

- find a pivot x to partition the list S into two sublists S_1 and S_2, such that $\forall y \in S_1 \ y < x$ and $\forall z \in S_2 \ z > x$

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

INPUT: set S of n elements and i;
OUTPUT: the ith smallest element in S;

Main idea:

- find a pivot x to partition the list S into two sublists S_1 and S_2, such that $\forall y \in S_1 \ y < x$ and $\forall z \in S_2 \ z > x$
- both S_1 and S_2 are guaranteed only a fraction of S;

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

INPUT: set S of n elements and i;
OUTPUT: the ith smallest element in S;

Main idea:

- find a pivot x to partition the list S into two sublists S_1 and S_2, such that $\forall y \in S_1 \; y < x$ and $\forall z \in S_2 \; z > x$
- both S_1 and S_2 are guaranteed only a fraction of S;
- the ith smallest element is either x, or in S_1 or in S_2 (but not both);

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

INPUT: set S of n elements and i;
OUTPUT: the ith smallest element in S;

Main idea:

- find a pivot x to partition the list S into two sublists S_1 and S_2, such that $\forall y \in S_1 \; y < x$ and $\forall z \in S_2 \; z > x$
- both S_1 and S_2 are guaranteed only a fraction of S;
- the ith smallest element is either x, or in S_1 or in S_2 (but not both);
- in either of the latter two cases, the algorithm is applied recursively.

Chapter 9. Medians and Order Statistics

Selection in worst case linear time

INPUT: set S of n elements and i;
OUTPUT: the ith smallest element in S;

Main idea:

- find a pivot x to partition the list S into two sublists S_1 and S_2, such that $\forall y \in S_1 \; y < x$ and $\forall z \in S_2 \; z > x$
- both S_1 and S_2 are guaranteed only a fraction of S;
- the ith smallest element is either x, or in S_1 or in S_2 (but not both);
- in either of the latter two cases, the algorithm is applied recursively.
- $T(n) \leq T(\beta n) + cn$ where $0 < \beta < 1$

Chapter 9. Medians and Order Statistics

How to find such a pivot?

How to find such a pivot?

- the very selection algorithm is recursively called for finding the pivot

Chapter 9. Medians and Order Statistics

How to find such a pivot?

- the very selection algorithm is recursively called for finding the pivot
- the size of the sublist to find the pivot is also a fraction αn

Chapter 9. Medians and Order Statistics

How to find such a pivot?

- the very selection algorithm is recursively called for finding the pivot
- the size of the sublist to find the pivot is also a fraction αn of the original list S, $|S| = n$;

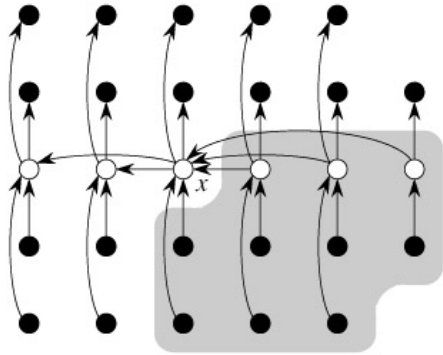
How to find such a pivot?

- the very selection algorithm is recursively called for finding the pivot
- the size of the sublist to find the pivot is also a fraction αn of the original list S, $|S| = n$;
- the total time actually is

$$T(n) \leq T(\alpha n) + T(\beta n) + cn$$

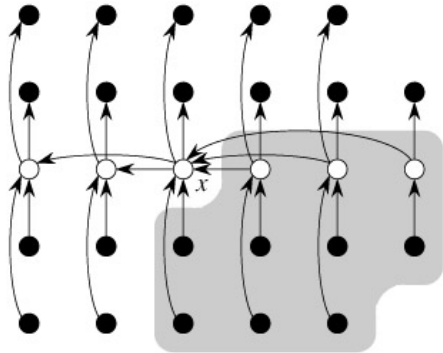
where $\alpha + \beta < 1$

Chapter 9. Medians and Order Statistics



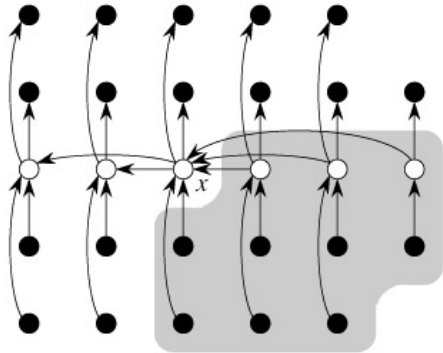
Algorithm SELECT (S, i); { where S contains n distinct elements}

Chapter 9. Medians and Order Statistics



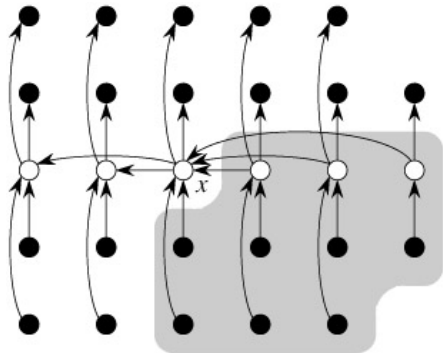
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements

Chapter 9. Medians and Order Statistics



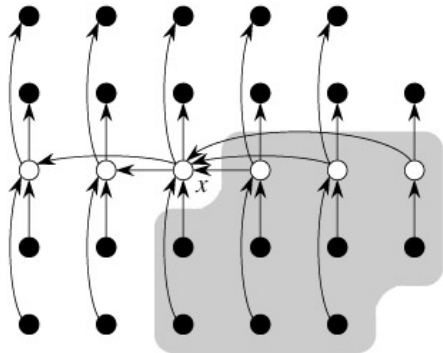
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;

Chapter 9. Medians and Order Statistics



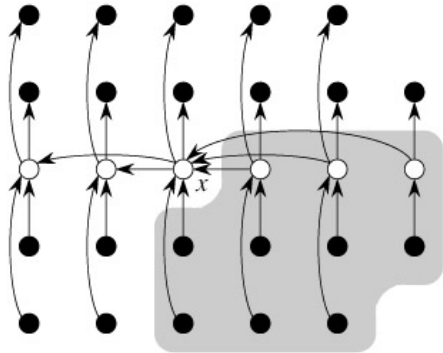
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
let M contain all these medians; where $|M| = \lceil n/5 \rceil$

Chapter 9. Medians and Order Statistics



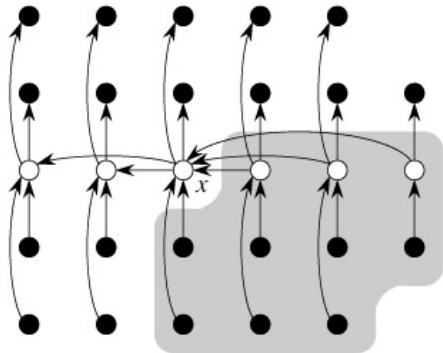
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let M contain all these medians; where $|M| = \lceil n/5 \rceil$
(3) **recursively call** SELECT$(M, \lceil n/10 \rceil)$;

Chapter 9. Medians and Order Statistics



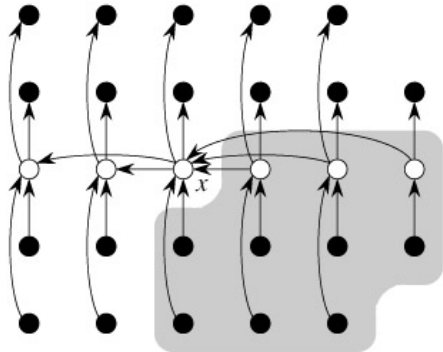
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let M contain all these medians; where $|M| = \lceil n/5 \rceil$
(3) **recursively call** SELECT$(M, \lceil n/10 \rceil)$;
    let the result be x and let the rank of x be k in S

Chapter 9. Medians and Order Statistics



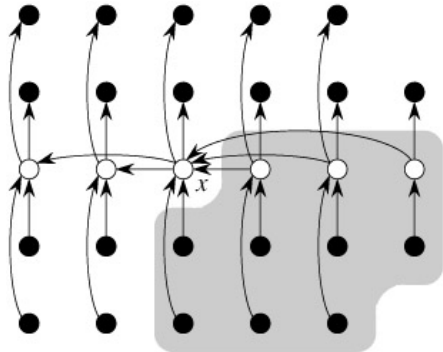
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let M contain all these medians; where $|M| = \lceil n/5 \rceil$
(3) **recursively call** SELECT$(M, \lceil n/10 \rceil)$;
    let the result be x and let the rank of x be k in S
(4) **if** $i = k$ **return** (x)

Chapter 9. Medians and Order Statistics



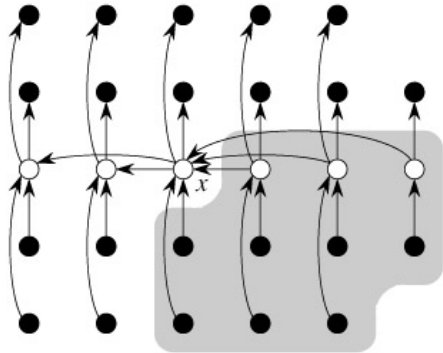
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let M contain all these medians; where $|M| = \lceil n/5 \rceil$
(3) **recursively call** SELECT$(M, \lceil n/10 \rceil)$;
    let the result be x and let the rank of x be k in S
(4) **if** $i = k$ **return** (x)
(5) **else** use x as the pivot to partition S resulting in S_1 and S_2,

Chapter 9. Medians and Order Statistics



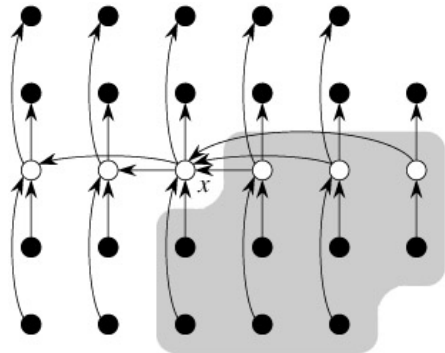
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let M contain all these medians; where $|M| = \lceil n/5 \rceil$
(3) **recursively call** SELECT$(M, \lceil n/10 \rceil)$;
    let the result be x and let the rank of x be k in S
(4) **if** $i = k$ **return** (x)
(5) **else** use x as the pivot to partition S resulting in S_1 and S_2,
    such that $\forall y \in S_1 \; y < x$ and $\forall z \in S_2 \; z > x$

Chapter 9. Medians and Order Statistics



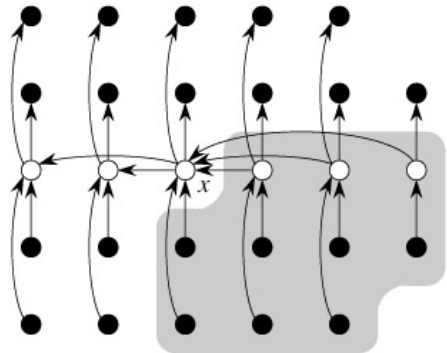
Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let M contain all these medians; where $|M| = \lceil n/5 \rceil$
(3) **recursively call** SELECT$(M, \lceil n/10 \rceil)$;
    let the result be x and let the rank of x be k in S
(4) **if** $i = k$ **return** (x)
(5) **else** use x as the pivot to partition S resulting in S_1 and S_2,
    such that $\forall y \in S_1 \; y < x$ and $\forall z \in S_2 \; z > x$
(6) **if** $i < k$ **recursively call** SELECT(S_1, i)

Chapter 9. Medians and Order Statistics



Algorithm SELECT (S, i); { where S contains n distinct elements}
(1) divide S into $\lceil n/5 \rceil$ groups of 5 elements
(2) sort each group (of 5) and find the median of each group;
    let M contain all these medians; where $|M| = \lceil n/5 \rceil$
(3) **recursively call** SELECT$(M, \lceil n/10 \rceil)$;
    let the result be x and let the rank of x be k in S
(4) **if** $i = k$ **return** (x)
(5) **else** use x as the pivot to partition S resulting in S_1 and S_2,
    such that $\forall y \in S_1 \ y < x$ and $\forall z \in S_2 \ z > x$
(6) **if** $i < k$ **recursively call** SELECT(S_1, i)
    else recursively call SELECT$(S_2, i - k)$

Chapter 9. Medians and Order Statistics



Note: the number of elements $\leq x$ is at least:

$$|S_1| = 3(\frac{\lceil n/5 \rceil}{2}) \geq 3n/10$$

similarly, the number of elements $\geq x$ is at least:

$$|S_2| \geq 3(\frac{\lceil n/5 \rceil}{2} - 2) \geq 3n/10 - 6$$

So a time upper bound for SELECT is

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 6 \rceil) + O(n)$$

when $n \geq 140$

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)
- prove the correctness of the bound.

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)
- prove the correctness of the bound.

For example, given Insertion Sort:

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)
- prove the correctness of the bound.

For example, given Insertion Sort:

- we first analyzed the algorithm

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )$)
- prove the correctness of the bound.

For example, given INSERTION SORT:

- we first analyzed the algorithm
  and obtained

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )\ $)
- prove the correctness of the bound.

For example, given INSERTION SORT:

- we first analyzed the algorithm
  and obtained

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$;

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Given an algorithm, carry out the following in order:

- analyzing time $T(n)$ of the algorithm
- obtain an expression $T(n) = \ldots$
- guess an upper (or lower) bound (e.g., $T(n) = O(\ )\ $)
- prove the correctness of the bound.

For example, given INSERTION SORT:

- we first analyzed the algorithm
  and obtained

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- we guessed upper bound $T(n) = O(n^2)$, i.e., $T(n) \leq cn^2$;
- and finally proved that it was indeed the case.

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function;

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function; or

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function; or

(2) we can use the substitution method by the principle of induction.

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function; or

(2) we can use the substitution method by the principle of induction.
But we need the recurrence to apply induction.

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function; or

(2) we can use the substitution method by the principle of induction.
But we need the recurrence to apply induction.

using the recurrence: $T(n) \leq T(\lfloor n/2 \rfloor) + c$

to prove $T(n) \leq c \log_2 n$.

Chapter 9. Medians and Order Statistics

Summary of Algorithm Analysis Scenarios

For recursive algorithms

For example, given BINARY SEARCH algorithm,

- we first analyze the time $T(n)$ of the algorithm and obtained a recurrence for $T(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

- we guess upper bound $T(n) = O(\log_2 n)$, i.e., $T(n) \leq c \log_2 n$;
- we prove the guessed bound.

(1) we can use the recursive tree method by unfolding the time function; or

(2) we can use the substitution method by the principle of induction.
But we need the recurrence to apply induction.

using the recurrence: $T(n) \leq T(\lfloor n/2 \rfloor) + c$

to prove $T(n) \leq c \log_2 n$. **see previous lecture notes**