Organization

1 Introduction

2 Sorting

3 Selection

Introduction

What is this Course About?

Clever ways to organize information in order to enable efficient computation

- What do we mean by clever?
- What do we mean by efficient?

Introduction

What is this Course About?

Clever ways to organize information in order to enable efficient computation

- What do we mean by clever?

  Using eficient data structure.

- What do we mean by efficient?

  To reduce time and space complexity, and
  to make the designing of algorithm easy.

Introduction

Data structure

- Lists, Stacks, Queues
- Heaps
- Binary Search Trees
- AVL Trees
- Hash Tables
- Graphs
- Disjoint Sets

Operations on data structure

- Insert
- Delete
- Find
- Merge
- Shortest Paths
- Union

Introduction

Applications: Used Everywhere!

- Systems
- Theory
- Graphics
- AI
- Other Applications

Perhaps the most important course in the CS curriculum!

Introduction

Goal of a Deterministic Algorithm

- The solution produced by the algorithm is correct, and
- the number of computational steps is same for different runs of the algorithm with the same input.

Introduction

Assymptotic Running time

- Worst case time complexity.
- Average case time complexity.
- Amortized time complexity

Introduction

Assymptotic Running time

- Worst case time complexity.
- Average case time complexity.
- Amortized time complexity

Worst case time complexity

An upper bound on the running time of the algorithm for any input.

Introduction

Assymptotic Running time

- Worst case time complexity.
- Average case time complexity.
- Amortized time complexity

Worst case time complexity

An upper bound on the running time of the algorithm for any input.

Average case time complexity

The average over the time required to run the algorithm for all possible inputs, where the input is assumed to follow some distribution (known in advance).

Introduction

Assymptotic Running time

- Worst case time complexity.
- Average case time complexity.
- Amortized time complexity

Worst case time complexity

An upper bound on the running time of the algorithm for any input.

Average case time complexity

The average over the time required to run the algorithm for all possible inputs, where the input is assumed to follow some distribution (known in advance).

Amortized time complexity

The total time needed for a series of input $\div$ number of runs

The time complexity is measured as a function of the input size, or the output size, or both.

Introduction

Specific goal of this lecture

- Becoming familiar with some of the fundamental data structures in computer science

Introduction

Specific goal of this lecture

- Becoming familiar with some of the fundamental data structures in computer science
- To improve the ability to solve problems abstractly
  – data structures are the building blocks

Introduction

Specific goal of this lecture

- Becoming familiar with some of the fundamental data structures in computer science
- To improve the ability to solve problems abstractly
  – data structures are the building blocks
- To improve the ability to analyze your algorithms
  – to prove correctness
  – analyzing the time complexity, and
  – improving the time complexity if possible

The Sorting Problem

Input:

A sequence of n numbers $a_1, a_2, \ldots, a_n$

Output:

A permutation (reordering) $a'_1, a'_2, \ldots, a'_n$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

The Sorting Problem

Input:

A sequence of n numbers $a_1, a_2, \ldots, a_n$

Output:

A permutation (reordering) $a'_1, a'_2, \ldots, a'_n$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Example

Before sorting: 70 85 12 56 42 37 5

After sorting: 5 12 37 42 56 70 85

Sorting

Structure of Data:

- Usually, the data to be sorted is a part of a collection of records
- Each record contains a key. The sorting of records is performed with respect to the values of the key.
- During the sorting, the keys are arranged along with the data attached with them.

Definitions

Internal Sort

The data to be sorted is all stored in the computer's main memory.

External Sort

Some of the data to be sorted might be stored in some external, slower, device.

In Place Sort

The amount of extra space required to sort the data is a small constant, and is not dependent of the input size.

STABLE sort

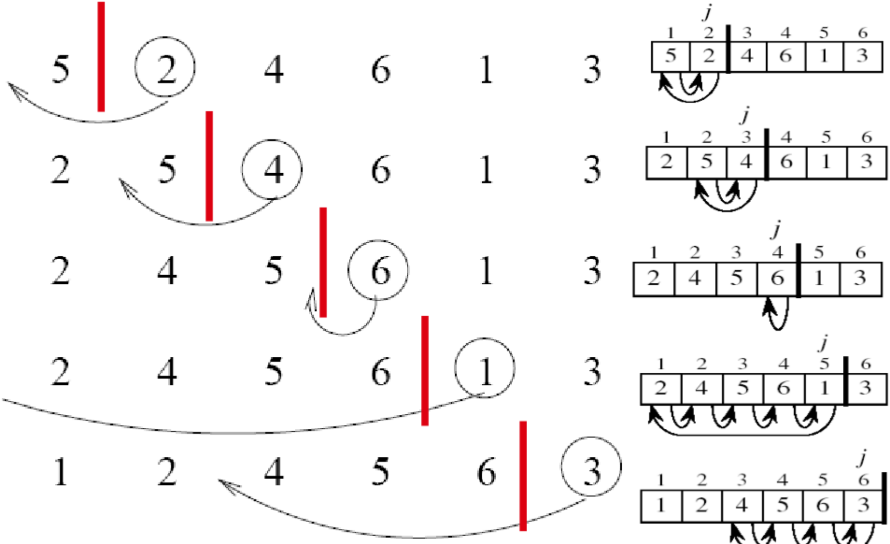
It preserves relative order of records with equal keys

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Insertion Sort

An example



Insertion Sort

Algorithm

Line 1 **for** $j \longleftarrow 2$ to n

Line 2 **do** $key = A[j]$

/* Insert $A[j]$ in the sorted list $A[1, 2, \ldots, j-1]$ */

Line 3 $i \longleftarrow j - 1$

Line 4 **while** $i > 0$ and $A[i] > key$

Line 5 **do** $A[i+1] \longleftarrow A[i]$

Line 6 $i \longleftarrow i - 1$

Line 7 $A[i+1] \longleftarrow key$

Insertion Sort

Algorithm

Line 1 **for** $j \longleftarrow 2$ to n
Line 2 **do** $key = A[j]$
/* Insert $A[j]$ in the sorted list $A[1, 2, \ldots, j-1]$ */
Line 3 $i \longleftarrow j - 1$
Line 4 **while** $i > 0$ and $A[i] > key$
Line 5 **do** $A[i+1] \longleftarrow A[i]$
Line 6 $i \longleftarrow i - 1$
Line 7 $A[i+1] \longleftarrow key$

Line number	1	2	3	4	5	6	7
times executed	$n-1$	$n-1$	$n-1$	$\sum_{j=2}^{n} t_j$	$\sum_{j=2}^{n}(t_j - 1)$	$\sum_{j=2}^{n}(t_j - 1)$	$n-1$

$t_j \longrightarrow$ number of times **while** statement is executed in iteration j.
$=$ number of elements greater than $A[j]$ in the input array
$= j - 1$ in the worst case

Analysis

Line number	1	2	3	4	5	6	7
times executed	$n-1$	$n-1$	$n-1$	$\sum_{j=2}^{n} t_j$	$\sum_{j=2}^{n}(t_j-1)$	$\sum_{j=2}^{n}(t_j-1)$	$n-1$

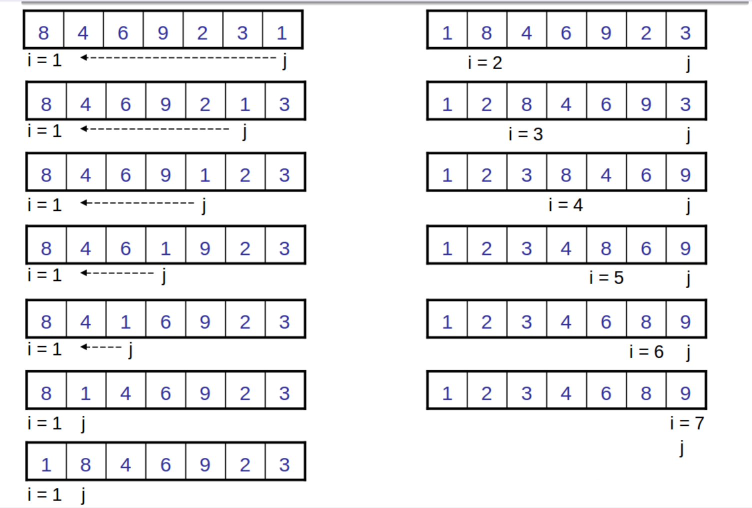
$t_j \longrightarrow$ number of times **while** statement is executed in iteration j.
$=$ number of elements greater than $A[j]$ in the input array
$= j-1$ in the worst case

Time Complexity

Time required to run the algorithm
$\leq 4(n-1) + 3\sum_{j=2}^{n} t_j$
$\leq kn^2$ for some constant k
$= O(n^2)$

Bubble Sort

An example



Algorithm

Algorithm

Line 1 **for** $i \longleftarrow 1$ to n /* $n = \text{length}(A)$ */

Line 2 **do for** $j = \text{length}(A)$ **downto** $i + 1$

/* fill $A[i]$ by $\min\{A[j], j = i, i+1, \ldots, n\}$ */

Line 3 **do if** $A[j] < A[j-1]$ **then** exchange$(A[j], A[j-1])$

Time Complexity: $O(n^2)$

Algorithm

Minor Improvement

Line 1 FLAG $=$ **false**; $i = 1$
Line 2 **while** FLAG $=$ **false do**
Line 3 FLAG $=$ **true**
Line 4 **for** $j = \text{length}(A)$ **downto** $i+1$ **do**
/* fill $A[i]$ by $\min\{A[j], j = i, i+1, \ldots, n\}$ */
Line 5 **if** $A[j] < A[j-1]$ **then**
Line 6 FLAG $=$ **false**
Line 7 exchange$(A[j], A[j-1])$
Line 8 $i = i + 1$

Algorithm

Minor Improvement

Line 1 FLAG $=$ **false**; $i = 1$
Line 2 **while** FLAG $=$ **false do**
Line 3 FLAG $=$ **true**
Line 4 **for** $j = \text{length}(A)$ **downto** $i + 1$ **do**
/* fill $A[i]$ by $\min\{A[j], j = i, i+1, \ldots, n\}$ */
Line 5 **if** $A[j] < A[j-1]$ **then**
Line 6 FLAG $=$ **false**
Line 7 exchange$(A[j], A[j-1])$
Line 8 $i = i + 1$

Time Complexity

Here iteration terminates if no exchange takes place in an iteration. Still worst case time complexity remains $O(n^2)$.

Algorithm

Minor Improvement

Line 1 FLAG $=$ **false**; $i = 1$
Line 2 **while** FLAG $=$ **false do**
Line 3 FLAG $=$ **true**
Line 4 **for** $j = \text{length}(A)$ **downto** $i + 1$ **do**
/* fill $A[i]$ by $\min\{A[j], j = i, i+1, \ldots, n\}$ */
Line 5 **if** $A[j] < A[j-1]$ **then**
Line 6 FLAG $=$ **false**
Line 7 exchange$(A[j], A[j-1])$
Line 8 $i = i + 1$

Time Complexity

Here iteration terminates if no exchange takes place in an iteration. Still worst case time complexity remains $O(n^2)$.

Example: 8 7 6 5 4 3 2 1

Quick Sort

QSORT(A, p, q)

1. If $p \geq q$, EXIT.
2. Compute $s \leftarrow$ correct position of $A[p]$ in the sorted order of the elements of A from p-th location to q-th location.
3. Move the pivot $A[p]$ into position $A[s]$.
4. Move the remaining elements of $A[p - q]$ into appropriate sides.
5. Recursively sort the segments to the left and right of the pivot.

5a. QSORT(A, p, $s - 1$);

5b. QSORT(A, $s + 1$, q).

Quick Sort - Detailed Algorithm

QSORT(A, p, q)

1. **if** $p \geq q$, EXIT.

2 & 4. Compute $j \leftarrow$ correct position of $A[p]$ in the sorted order of the elements of A from p-th location to q-th location.

2a. pivot $= A[p]; i = p + 1;\ j = q$

2b. **while** $(i < j)$ **do**

2c. **while** $A[i] \leq$ pivot **do** $i = i + 1$

2d. **while** $A[j] >$ pivot **do** $j = j - 1$

2e. **if** $i < j$ **then** SWAP $(A[i], A[j])$

3. Move the pivot $A[p]$ into position $A[s]$.

3a. SWAP$(A[p], A[j])$

5. Recursively sort the segments to the left and right of the pivot.

5a. QSORT(A, p, $s - 1$);

5b. QSORT(A, $s + 1$, q).

Complexity Results of QSORT

- An INPLACE algorithm
- The worst case time complexity is $O(n^2)$.
- The average case time complexity is $O(n \log n)$.

Complexity Results of QSORT

- An INPLACE algorithm
- The worst case time complexity is $O(n^2)$.
- The average case time complexity is $O(n \log n)$.

Worst case analysis

- In each level of recursion, total number of comparisons $= O(n)$.
- Number of levels of recursion $= O(n)$ in the worst case.

Total time: $O(n^2)$ in the worst case.

Complexity Results of QSORT

- An INPLACE algorithm
- The worst case time complexity is $O(n^2)$.
- The average case time complexity is $O(n \log n)$.

Worst case analysis

- In each level of recursion, total number of comparisons $= O(n)$.
- Number of levels of recursion $= O(n)$ in the worst case.

Total time: $O(n^2)$ in the worst case.

Example:

array	pivot
[3, 1, 4, 4, 8, 2, 7]	3
[3, 1, 2, 4, 8, 4, 7]	
[2, 1], 3, [4, 4, 8, 2, 7]	2
1, 2, 3, [4, 4, 8, 7]	4
1, 2, 3, 4, [4, 8, 8]	4
1, 2, 3, 4, 4, [8, 7]	8
1, 2, 3, 4, 4, [7], 8	7
1, 2, 3, 4, 4, 7, 8	

Average Case Analysis of Quick Sort

$T(n) \longrightarrow$ Expected time taken by the QSORT algorithm on an input of size n.

Inductive proof of $T(n) = O(n \log n)$

Average Case Analysis of Quick Sort

$T(n) \longrightarrow$ Expected time taken by the QSORT algorithm on an input of size n.

Inductive proof of $T(n) = O(n \log n)$

- Assumption: pivot can lie anywhere in $[1, 2, \ldots, n]$ of the input array with equal probability.

Average Case Analysis of Quick Sort

$T(n) \longrightarrow$ Expected time taken by the QSORT algorithm on an input of size n.

Inductive proof of $T(n) = O(n \log n)$

- Assumption: pivot can lie anywhere in $[1, 2, \ldots, n]$ of the input array with equal probability.
- $T(n) = \frac{1}{n}[T(1) + T(n) + \sum_{q=1}^{n-1}(T(q) + T(n-q))] + O(n)$.
- $\frac{1}{n}(T(1) + T(n-1)) = O(n)$ since $T(1) = 1$ and $T(n-1) = O(n^2)$.

Average Case Analysis of Quick Sort

$T(n) \longrightarrow$ Expected time taken by the QSORT algorithm on an input of size n.

Inductive proof of $T(n) = O(n \log n)$

- Assumption: pivot can lie anywhere in $[1, 2, \ldots, n]$ of the input array with equal probability.
- $T(n) = \frac{1}{n}[T(1) + T(n) + \sum_{q=1}^{n-1}(T(q) + T(n-q))] + O(n)$.
- $\frac{1}{n}(T(1) + T(n-1)) = O(n)$ since $T(1) = 1$ and $T(n-1) = O(n^2)$.
- $T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + O(n) \leq \frac{2}{n}\sum_{k=1}^{n-1}(ak \log k + b) + O(n)$

  $= \frac{2a}{n}\sum_{k=1}^{n-1} k \log k + \frac{2b}{n}(n-1) + O(n)$

Average Case Analysis of Quick Sort

$T(n) \longrightarrow$ Expected time taken by the QSORT algorithm on an input of size n.

Inductive proof of $T(n) = O(n \log n)$

- Assumption: pivot can lie anywhere in $[1, 2, \ldots, n]$ of the input array with equal probability.
- $T(n) = \frac{1}{n}[T(1) + T(n) + \sum_{q=1}^{n-1}(T(q) + T(n-q))] + O(n)$.
- $\frac{1}{n}(T(1) + T(n-1)) = O(n)$ since $T(1) = 1$ and $T(n-1) = O(n^2)$.
- $T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + O(n) \leq \frac{2}{n}\sum_{k=1}^{n-1}(ak \log k + b) + O(n)$

  $= \frac{2a}{n}\sum_{k=1}^{n-1} k \log k + \frac{2b}{n}(n-1) + O(n)$
- Using $\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$, we have
- $T(n) = \frac{2a}{n}(\frac{n^2 \log n}{2} - \frac{n^2}{8}) + \frac{2b}{n}(n-1) + O(n)$

Average Case Analysis of Quick Sort

$T(n) \longrightarrow$ Expected time taken by the QSORT algorithm on an input of size n.

Inductive proof of $T(n) = O(n \log n)$

- Assumption: pivot can lie anywhere in $[1, 2, \ldots, n]$ of the input array with equal probability.
- $T(n) = \frac{1}{n}[T(1) + T(n) + \sum_{q=1}^{n-1}(T(q) + T(n-q))] + O(n)$.
- $\frac{1}{n}(T(1) + T(n-1)) = O(n)$ since $T(1) = 1$ and $T(n-1) = O(n^2)$.
- $T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + O(n) \leq \frac{2}{n}\sum_{k=1}^{n-1}(ak \log k + b) + O(n)$

  $= \frac{2a}{n}\sum_{k=1}^{n-1} k \log k + \frac{2b}{n}(n-1) + O(n)$
- Using $\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$, we have
- $T(n) = \frac{2a}{n}(\frac{n^2 \log n}{2} - \frac{n^2}{8}) + \frac{2b}{n}(n-1) + O(n)$

  $= an \log n - \frac{1}{4}an + 2b + O(n) \leq an \log n + b$
  (since we can choose a large enough such that $\frac{an}{4} > O(n) + b$)

Proof of $\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$

$\sum_{k=1}^{n-1} k \log k \leq (\log n - 1) \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log n \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k$

[Reason: $k \log k \leq k \log \frac{n}{2}$ for $k \leq \frac{n}{2}$, and $k \log k \leq k \log n$ for $k \geq \frac{n}{2}$.]

$= \log n \sum_{k-1}^{n-1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k$

$= \frac{n(n-1)}{2} \log n - \frac{1}{2}(\frac{n}{2} - 1)\frac{n}{2} \leq \frac{n^2 \log n}{2} - \frac{n^2}{8}$

Scope of improvement

Observations:

- In each level of recursion, total number of comparisons $= O(n)$.
- Thus, we need to reduce the number of levels of recursion.

Scope of improvement

Observations:

- In each level of recursion, total number of comparisons = $O(n)$.
- Thus, we need to reduce the number of levels of recursion.

Avenue: Split each sub-array under consideration, into two equal halves using the median of that sub-array as the *pivot*.

Scope of improvement

Observations:

- In each level of recursion, total number of comparisons $= O(n)$.
- Thus, we need to reduce the number of levels of recursion.

Avenue: Split each sub-array under consideration, into two equal halves using the median of that sub-array as the *pivot*.
Reason: If there are k levels then $2^k = n$
$\implies k = \log_2 n$.

Scope of improvement

Observations:

- In each level of recursion, total number of comparisons $= O(n)$.
- Thus, we need to reduce the number of levels of recursion.

Avenue: Split each sub-array under consideration, into two equal halves using the median of that sub-array as the *pivot*.
Reason: If there are k levels then $2^k = n$
$\implies k = \log_2 n$.
Tool: Median of an array of n elements can be computed in $O(n)$ time.

Scope of improvement

Observations:

- In each level of recursion, total number of comparisons $= O(n)$.
- Thus, we need to reduce the number of levels of recursion.

Avenue: Split each sub-array under consideration, into two equal halves using the median of that sub-array as the *pivot*.
Reason: If there are k levels then $2^k = n$
$\implies k = \log_2 n$.
Tool: Median of an array of n elements can be computed in $O(n)$ time.

Time Complexity of the algorithm is $O(n \log_2 n)$.

Scope of improvement

Observations:

- In each level of recursion, total number of comparisons $= O(n)$.
- Thus, we need to reduce the number of levels of recursion.

Avenue: Split each sub-array under consideration, into two equal halves using the median of that sub-array as the *pivot*.
Reason: If there are k levels then $2^k = n$
$\implies k = \log_2 n$.
Tool: Median of an array of n elements can be computed in $O(n)$ time.

Time Complexity of the algorithm is $O(n \log_2 n)$.

Difficulty: The linear time algorithm for computing median is difficult to implement.

Randomized Quick Sort

An Useful Concept - The Central Splitter

It is an index s such that the number of elements less (resp. greater) than $A[s]$ is at least $\frac{n}{4}$.

Randomized Quick Sort

An Useful Concept - The Central Splitter

It is an index s such that the number of elements less (resp. greater) than $A[s]$ is at least $\frac{n}{4}$.

- The algorithm randomly chooses a key, and checks whether it is a central splitter or not.
- If it is a central splitter, then the array is split with that key as was done in the QSORT algorithm.
- It can be shown that the expected number of trials needed to get a central splitter is constant.

Randomized Quick Sort

RandQSORT(A, p, q)

1: If $p \geq q$, then EXIT.

2: While no central splitter has been found, execute the following steps:

  2.1: Choose uniformly at random a number $r \in \{p, p+1, \ldots, q\}$.

  2.2: Compute s = number of elements in A that are less than $A[r]$, and
  t = number of elements in A that are greater than $A[r]$.

  2.3: If $s \geq \frac{q-p}{4}$ and $t \geq \frac{q-p}{4}$, then $A[r]$ is a central splitter.

3: Position $A[r]$ in $A[s+1]$, put the members in A that are smaller than the central splitter in $A[p \ldots s]$ and the members in A that are larger than the central splitter in $A[s+2 \ldots q]$.

4: RandQSORT(A, p, s);

5: RandQSORT(A, $s+2$, q).

Analysis of RandQSORT

Fact: Step 2 needs $O(q-p)$ time.

Question: How many times Step 2 is executed for finding a central splitter ?

Analysis of RandQSORT

Fact: Step 2 needs $O(q - p)$ time.

Question: How many times Step 2 is executed for finding a central splitter ?

Result:

The probability that the randomly chosen element is a central splitter is $\frac{1}{2}$.

Analysis of RandQSORT

Fact: Step 2 needs $O(q-p)$ time.

Question: How many times Step 2 is executed for finding a central splitter ?

Result:

The probability that the randomly chosen element is a central splitter is $\frac{1}{2}$.

Implication

- The expected number of times the Step 2 needs to be repeated to get a central splitter is 2.
- Thus, the expected time complexity of Step 2 is $O(n)$

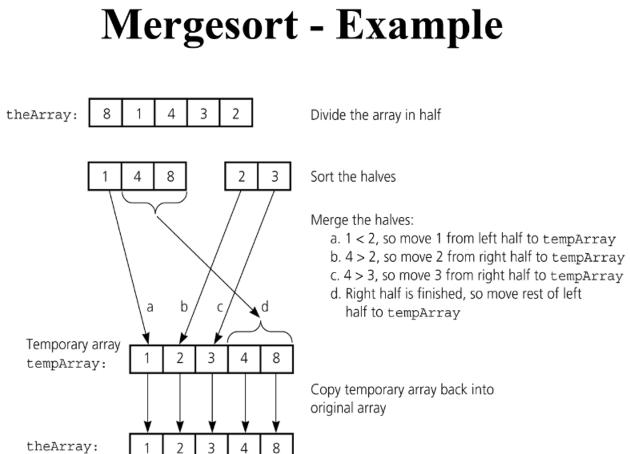
Analysis of RandQSORT

Time Complexity

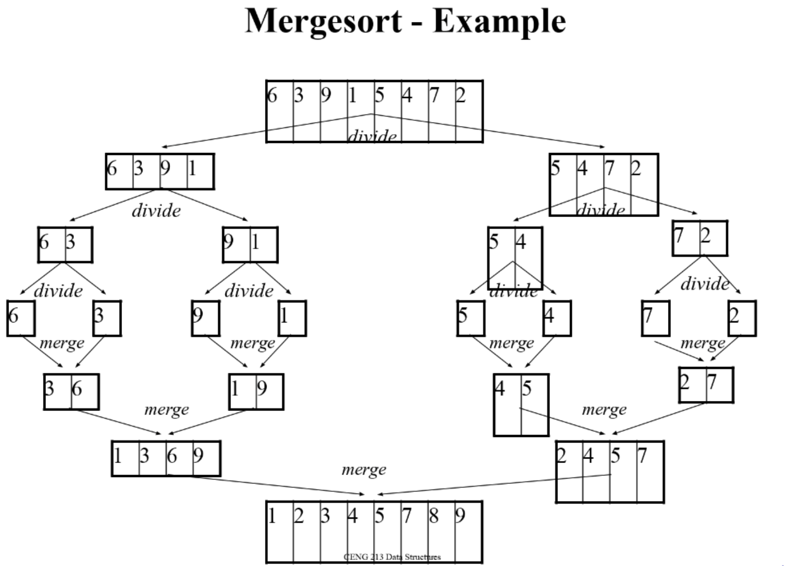
- Worst case size of each partition in j-th level of recursion is $n \times (\frac{3}{4})^j$.
- Number of levels of recursion $= \log_{\frac{4}{3}} n = O(\log n)$.
- Recurrence Relation of the time complexity: $T(n) = 2T(\frac{3n}{4}) + O(n) = O(n \log n)$

Merge Sort - A divide and conquer algorithm

- Divide the list into two halves
- Sort each half separately
- Merge the sorted halves into one sorted array



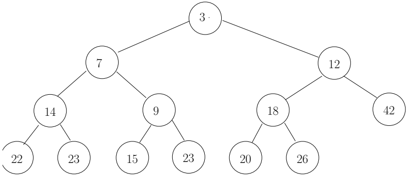
Merge Sort


Mergesort - Example
6 3 9 1 5 4 7 2
divide
6 3 9 1
5 4 7 2
divide
divide
6 3
9 1
5 4
7 2
divide
divide
divide
divide
6
3
9
1
5
4
7
2
merge
merge
merge
merge
3 6
1 9
4 5
2 7
merge
merge
1 3 6 9
2 4 5 7
merge
1 2 3 4 5 7 8 9
CENG 213 Data Structures

Heap

Definition

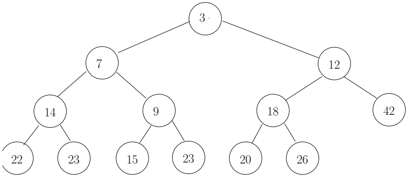
A heap is a complete binary tree with elements from a partially ordered set, such that the element at every node is less than (or equal to) the elements in the subtree rooted at that node.



Heap

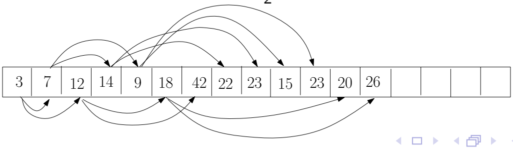
Definition

A heap is a complete binary tree with elements from a partially ordered set, such that the element at every node is less than (or equal to) the elements in the subtree rooted at that node.



As it is a complete binary tree, it can be represented in an array, where

- *root* is at location 1.
- The children of node i (if exists) are at locations $2i$ and $2i+1$.
- The parent of a node i is at location $\frac{i}{2}$



Heap

- Because of its structure, a heap of height k will have between 2^k and $2^{k+1}1$ elements. Therefore a heap with n elements will have height $= \lfloor \log_2 n \rfloor$.
- Because of the heap property, the minimum element will always be present at the root of the heap. Thus the *findmin* operation will have worst-case $O(1)$ running time.

Heap

- Because of its structure, a heap of height k will have between 2^k and $2^{k+1}1$ elements. Therefore a heap with n elements will have height $= \lfloor \log_2 n \rfloor$.
- Because of the heap property, the minimum element will always be present at the root of the heap. Thus the *findmin* operation will have worst-case $O(1)$ running time.

Heap can be used to implement priority queue

The insert routine in priority queue (heap) H is as follows:

- Let n be the size of the heap.
- Insert the new element x in position $H[n+1]$.
- Adjust the location of $H[n+1]$
  - Let i be the current location of x (* initially $i = n+1$ *)
  - while not $(H[i] \geq H[\frac{i}{2}])$ or $(i = 1)$ do swap($H[i]$,$H[\frac{i}{2}]$)

Heap

- Because of its structure, a heap of height k will have between 2^k and $2^{k+1}1$ elements. Therefore a heap with n elements will have height $= \lfloor \log_2 n \rfloor$.
- Because of the heap property, the minimum element will always be present at the root of the heap. Thus the *findmin* operation will have worst-case $O(1)$ running time.

Heap can be used to implement priority queue

The insert routine in priority queue (heap) H is as follows:

- Let n be the size of the heap.
- Insert the new element x in position $H[n+1]$.
- Adjust the location of $H[n+1]$
  - Let i be the current location of x (* initially $i = n+1$ *)
  - while not $(H[i] \geq H[\frac{i}{2}])$ or $(i = 1)$ do swap$(H[i], H[\frac{i}{2}])$

Priority queue (implemented as a heap) can be maintained in $O(\log n)$ time, where n is the maximum size of the heap.

Heap Sort

Heapify$(A,\ i)$

Line 1: $\ell \longleftarrow \text{left}(i) = 2i$; $r \longleftarrow \text{right}(i) = 2i + 1$

Line 2: **if** $\ell \leq n$ and $A[\ell] > A[i]$ **then** largest $\longleftarrow \ell$ **else** largest $\longleftarrow r$

Line 3: **if** $r \leq n$ and $A[r] > A[\text{largest}]$ **then** largest $\longleftarrow r$

Line 4: **if** largest $\neq i$ **then** SWAP$(A[i], A[\text{largest}])$; Heapify$(A, \text{largest})$

BuildHeap(A)

Line 1: **for** $i = \lfloor \frac{n}{2} \rfloor$downto1

Line 2: Heapify(A, i)

Analysis of Heapify

Trivial Analysis

- Time required for settling relation between $A[i]$, $A[2i]$ and $A[2i+1]$ is $O(1)$
- $T(n) \longrightarrow$ Time complexity for running Heapify for $A[1]$
- Maximum number of children of one child of $A[1]$ is $\frac{2n}{3}$
- Thus, $T(n) = T(\frac{2n}{3}) + O(1) = O(\log n)$

Time complexity for BuildHeap is $O(n \log n)$.

Analysis of Heapify

Tighter Analysis

Fact: In an n element heap, the number of nodes at height h is at most $\lceil \frac{n}{2^{h+1}} \rceil$.

- Time complexity for BuildHeap
  $= \sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil (h)$
  $= O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}) = 2n$

Heapsort

HeapSort(A)

Line 1: **for** $i = \lfloor \frac{n}{2} \rfloor$downto1 **do** Heapify (A, i)

Line 2: **for** $m = n$downto2 **do**

SWAP$(A[i], A[m])$

if $m > 2$ **then** Heapify$(A[1, 2, \ldots, m-1], 1)$

Heapsort

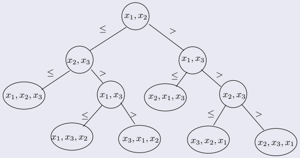
Analysis

- Time for deletion of a node from a Heap containing k nodes is at most $2\lfloor \log k \rfloor$
- So, the total time complexity $= 2\sum_{k=1}^{n-1} \lfloor \log k \rfloor$
- $\sum_{k=1}^{n-1} \lfloor \log k \rfloor = \underline{\log 2 + \log 3} + \underline{\log 4 + \log 5 + \log 6 + \log 7} + \ldots$
  $\leq 2\log 2 + 4\log 4 + \ldots 2^{\lfloor \log n \rfloor - 1}(\lfloor \log n \rfloor - 1) + \lfloor \log n \rfloor + (\lfloor \log n \rfloor + 1) + \ldots + \lfloor \log(n-1) \rfloor$
  $\leq \sum_{k=1}^{\lfloor \log n \rfloor - 1} k2^k + \lfloor \log n \rfloor (n - 2^{\lfloor \log n \rfloor})$
- $\sum_{k=1}^{\lfloor \log n \rfloor} k2^k = (\lfloor \log n \rfloor - 2)2^{\lceil \log n \rceil} + 2$
- substituting, we have
  Total time required $= 2n\log n - 4n + 4 = O(n\log n)$

Lower Bound for Sorting

Decision tree for Bubble Sort

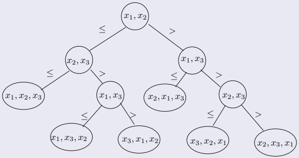
Given string x_1, x_2, x_3



Lower Bound for Sorting

Decision tree for Bubble Sort

Given string x_1, x_2, x_3



Result

Let ℓ be the number of leaves in a binary tree, and d be the depth of the tree, then $\ell \leq 2^d$, i.e., $d \geq \lceil \log_2 \ell \rceil$

In our case, $\ell = n!$. Thus $d \geq \lceil \log_2 n! \rceil$

Lower Bound for Sorting

Any algorithm to sort n items by comparison of keys must do at least $\lceil \log_2 n! \rceil$, or approximately $\lceil n \log_2 n - 1.5n \rceil$ comparisons in the worst case.

Proof:

$$
\begin{aligned}
\log_2 n! &= \textstyle\sum_{j=1}^{n} \log_2 j \\
&\geq \textstyle\int_1^n \log_2 x dx \\
&= \log_2 e \textstyle\int_1^n \log_e x dx \\
&= \log_2 e[x \log_e x - x]_1^n \\
&= n \log_2 n - 1.44n \text{ (Putting } \log_2 e = 1.44\text{)}
\end{aligned}
$$

Lower Bound for Sorting - more stronger result

Average number of comparisons done by an algorithm to sort n items by comparison of keys is at least
$\lfloor \log_2 n! \rfloor = \lfloor n \log_2 n - 1.5m \rfloor$

Proof: Average number of comparisons = Average path length in the decision tree with $n!$ leaves.

$$= \frac{\text{Total path length of all leaves in the decision tree}}{n!} = \frac{n!\lfloor \log_2 n! \rfloor + (n! - 2^{\lfloor \log_2 n! \rfloor})}{n!}$$

$= \lfloor \log n! \rfloor + \epsilon$, where $0 < \epsilon < 1$ since $n! - 2^{\lfloor \log_2 n! \rfloor} < \frac{n!}{2}$

Lower Bound for Sorting - more stronger result

The minimum of the sum of path lengths with ℓ leaves is $\ell\lfloor\log_2 \ell\rfloor + 2(\ell - 2^{\lfloor\log_2 \ell\rfloor})$

Proof:

- If ℓ is a power of 2, then all the leaves are in level $\log \ell$. Thus, the sum of path lengths is $\ell \log_2 \ell$.
- If ℓ is not a power of 2, then the depth of the tree is $d = \lceil \log_2 \ell \rceil$, and the leaves are in d-th and $d-1$-th level.
- The sum of length of all paths up to $d-1$-th level is $\ell(d-1)$.
- For each leaf in level d, 1 is to be added.
- Note that, at level $d-1$, k_1 nodes have two children and k_2 nodes have no children. Thus, $k_1 + k_2 = 2^{d-1}$ and $2k_1 + k_2 = \ell$.
- Thus, the number of leaves in level d is $2(\ell - 2^{d-1})$.
- Thus, the sum of path lengths = $\ell(d-1) + 2(\ell - 2^{d-1}) = \ell\lfloor\log_2 \ell\rfloor + 2(\ell - 2^{\lfloor\log_2 \ell\rfloor})$.

Counting Sort

Input: A sequence A of n integers, each one lies in $\{1, 2, \ldots, m\}$.
Output: The sequence A in sorted order.
Algorithm

Step 1: In a linear scan compute frequency f_i of each number $i \in \{1, 2, \ldots, m\}$

Step 2: Write the number i f_i times in the array A starting from its position 1. This needs $O(n)$ time.

Counting Sort

Input: A sequence A of n integers, each one lies in $\{1, 2, \ldots, m\}$.
Output: The sequence A in sorted order.
Algorithm

Step 1: In a linear scan compute frequency f_i of each number $i \in \{1, 2, \ldots, m\}$

Step 2: Write the number i f_i times in the array A starting from its position 1. This needs $O(n)$ time.

Fallacy

Question: Does it violate the lower bound of the sorting problem?
Answer: No. The reason is that the computational model for sorting is different.

Sorting in Read-only memory

Input data resides in a read-only array
Output is produced in some other output device

Trivial algorithm — $O(n^2)$

Sorting in Read-only memory

Input data resides in a read-only array
Output is produced in some other output device

Trivial algorithm — $O(n^2)$

With k space

- The whole array is split into k equal parts. start and end indices of each part is stored in an array.
- Initially, a min-heap is formed with the minimum element in each part. with each element, its part number is noted.
  $---$ $O(n)$ time.
- The minimum element is reported, and deleted from the heap. The minimum element of the corresponding part is obtained by a linear scan in that part, and is put in the heap.
  $---$ $O(\frac{n}{k})$ time.

Sorting in Read-only memory

Input data resides in a read-only array
Output is produced in some other output device

Trivial algorithm — $O(n^2)$

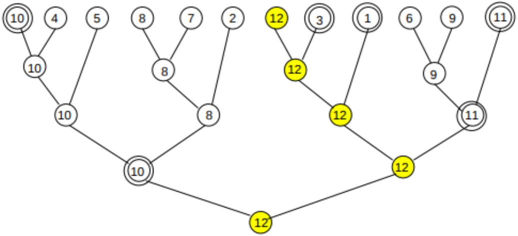
With k space

- The whole array is split into k equal parts. start and end indices of each part is stored in an array.
- Initially, a min-heap is formed with the minimum element in each part. with each element, its part number is noted.
  $- - -$ $O(n)$ time.
- The minimum element is reported, and deleted from the heap. The minimum element of the corresponding part is obtained by a linear scan in that part, and is put in the heap.
  $- - -$ $O(\frac{n}{k})$ time.

The overall time complexity is $O(\frac{n^2}{k})$ with $O(k)$ space

Finding extreme elements

- Finding the smallest number needs $n - 1$ comparisons.
- Finding smallest and largest numbers need $\frac{3n}{2}$ comparisons.
- Finding smallest and second smallest numbers need $n + \log_2 n - 2$ comparisons



Median Finding

The Problem

Given an array $A[1 \dots n]$ containing n (comparable) elements, find an element $A[i]$ such that
No. of elements smaller than $A[i]$ = No. of elements greater than $A[i] = \lfloor \frac{n}{2} \rfloor$.

Median Finding

The Problem

Given an array $A[1 \ldots n]$ containing n (comparable) elements, find an element $A[i]$ such that
No. of elements smaller than $A[i]$ = No. of elements greater than $A[i] = \lfloor \frac{n}{2} \rfloor$.

Trivial Algorithm

Sort the members in the array A, and report the middle-most element.
Time complexity: $O(n \log n)$

Median Finding

The Problem

Given an array $A[1 \dots n]$ containing n (comparable) elements, find an element $A[i]$ such that
No. of elements smaller than $A[i]$ = No. of elements greater than $A[i]$ = $\lfloor \frac{n}{2} \rfloor$.

Trivial Algorithm

Sort the members in the array A, and report the middle-most element.
Time complexity: $O(n \log n)$

Target

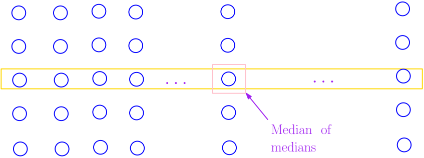
A linear time algorithm.

Deterministic Algorithm

Prune-and-Search Algorithm

function SELECT(A, k)
Output: The k-th smallest element in the input array A
if $|A| \leq 20$ **then**
  $x =$ k-th smallest element in A obtained by sorting the members in A
  return x
Split the array A into $\frac{|A|}{5}$ parts each of size 5.
For each part, compute the median, and store them in an array B
$x = \text{SELECT}(B, \lceil \frac{|B|}{2} \rceil)$
Split A into the following three parts:
  L: elements less than x
  E: elements equal to x
  G: elements greater than x
if $k \leq |L|$ **then** SELECT$(L.k)$
  else if $k \leq |L| + |E|$ **then return** x
  else SELECT$(G,\ k - |L| - |E|)$

Time Complexity



Recurrence relation

$$T(n) = T(\tfrac{n}{5}) + T(\tfrac{3n}{4}) + cn$$

Assuming $T(n) = kn$, we have $\frac{kn}{20} = cn$,
Implying $k = 20c$.
Thus, when $n \leq 20$, we stopped recursion, and applied bruteforce algorithm.

Randomized Version

Splitter based Algorithm

function Rand_SELECT(A, k)
central_splitter = NO
while not *central_splitter* **do**
    Choose an element $A[i]$ randomly
    for $j = 1, \ldots, n$ **do**
        if $A[j] < A[i]$ put $A[j]$ in L
        if $A[j] > A[i]$ put $A[j]$ in G
    endfor
    if $|L| \geq \frac{n}{4}$ and $|G| \geq \frac{n}{4}$ **then**
        central_splitter = YES
    endif
endwhile
if $|L| = k - 1$ **then return** $A[i]$
    else if $|L| > k$ **then** Rand_SELECT(L, k)
    else Rand_SELECT(G, $k - 1 - |L|$)

Problem: Choosing a good splitter.

Splitter Selection

Arbitrary splitter

Here if always the splitter is far from median, then in the worst case time required may be
$T(n) \leq cn + c(n-1) + \ldots = \frac{cn(n-1)}{2} = \Theta(n^2)$.

Splitter Selection

Arbitrary splitter

Here if always the splitter is far from median, then in the worst case time required may be
$T(n) \leq cn + c(n-1) + \ldots = \frac{cn(n-1)}{2} = \Theta(n^2)$.

Good splitter

It is an index s such that the number of elements less (resp. greater) than $A[s]$ is at least $\epsilon \times n$.

Splitter Selection

Arbitrary splitter

Here if always the splitter is far from median, then in the worst case time required may be
$T(n) \leq cn + c(n-1) + \ldots = \frac{cn(n-1)}{2} = \Theta(n^2)$.

Good splitter

It is an index s such that the number of elements less (resp. greater) than $A[s]$ is at least $\epsilon \times n$.

- A good splitter assures that at least a constant fraction ϵ of the total points must be thrown out in each iteration,
- Thus, in the worst case time complexity becomes
  $T(n) \leq T((1-\epsilon))n) + cn = O(n)$.

Splitter Selection

Arbitrary splitter

Here if always the splitter is far from median, then in the worst case time required may be
$T(n) \leq cn + c(n-1) + \ldots = \frac{cn(n-1)}{2} = \Theta(n^2)$.

Good splitter

It is an index s such that the number of elements less (resp. greater) than $A[s]$ is at least $\epsilon \times n$.

- A good splitter assures that at least a constant fraction ϵ of the total points must be thrown out in each iteration,
- Thus, in the worst case time complexity becomes $T(n) \leq T((1-\epsilon))n) + cn = O(n)$.

Remark: In the earlier algorithm, it is assured that in the next iteration at least $\frac{n}{10}$ will be thrown out.

Randomized Splitter

- Choosing $\epsilon = \frac{1}{4}$, we have probability of getting a *good splitter* in a random trial $= \frac{1}{2}$.
- Number of comparisons required to check whether a chosen splitter is good is n.
- Thus, expected time to get a good splitter is $2cn$.

Randomized Splitter

In each iteration, we are deleting at least $\frac{1}{4}$ fraction of points. Thus in the j-th iteration, we have $(\frac{3}{4})^j n$ points.

Expected time complexity

- X_j - Number of steps executed in the j-th phase.
- Total execution time $= X = X_0 + X_1 + X_2 + \ldots$
- By linearity of expectation, we have $E(X) = E(X_0) + E(X_1) + E(X_2) + \ldots$.
- $E(X_j) = 2cn(\frac{3}{4})^j$

Expected Time Complexity $= 8cn$

A nice problem

Suppose a given set of data in an array is organized as a MAX-HEAP.

Problem: To search for minimum and next minimum.

Question: In what portion of the array, we need to search to answer the query.