CSE 548: Algorithms

Fall 2022

R. Sekar

Divide-and-Conquer: A versatile strategy

Steps

- Break a problem into subproblems that are smaller instances of the same problem
- Recursively solve these subproblems
- Combine these answers to obtain the solution to the problem

Divide-and-Conquer: A versatile strategy

Steps

- Break a problem into subproblems that are smaller instances of the same problem
- Recursively solve these subproblems
- Combine these answers to obtain the solution to the problem

Benefits

Conceptual simplification

Speed up:

- rapidly (exponentially) reduce problem space
- exploit commonalities in subproblem solutions

Parallelism: Divide-and-conquer algorithms are amenable to parallelization

Locality: Their depth-first nature increases locality, extremely important for today's processors.

Topics

1. Warmup
   - Overview
   - Search
   - Exponentiation
2. Sorting
   - Mergesort
   - Recurrences
   - Quicksort
   - Lower Bound
   - Radix sort
3. Selection
   - Select k-th min
   - Priority Queues
4. Multiplication
   - Matrix Multiplication
   - Integer multiplication

Binary Search

Problem: Find a key k in an ordered collection

Binary Search

Problem: Find a key k in an ordered collection

Examples: Sorted array $A[n]$: Compare k with $A[n/2]$, then recursively search in $A[0 \cdots (n/2-1)]$ (if $k < A[n/2]$) or $A[n/2 \cdots n]$ (otherwise)

Binary Search

Problem: Find a key k in an ordered collection

Examples: Sorted array $A[n]$: Compare k with $A[n/2]$, then recursively search in $A[0 \cdots (n/2-1)]$ (if $k < A[n/2]$) or $A[n/2 \cdots n]$ (otherwise)

Binary search tree T: Compare k with $root(T)$, based on the result, recursively search left or right subtree of root.

Binary Search

Problem: Find a key k in an ordered collection

Examples: Sorted array $A[n]$: Compare k with $A[n/2]$, then recursively search in $A[0 \cdots (n/2 - 1)]$ (if $k < A[n/2]$) or $A[n/2 \cdots n]$ (otherwise)

Binary search tree T: Compare k with $root(T)$, based on the result, recursively search left or right subtree of root.

B-Tree: Hybrid of the above two. Root stores an array M of m keys, and has $m + 1$ children. Use binary search on M to identify which child can contain k, recursively search that subtree.

Exponentiation

- How many multiplications are required to compute x^n?

Exponentiation

- How many multiplications are required to compute x^n?
- Can we use a divide-and-conquer approach to make it faster?

Exponentiation

- How many multiplications are required to compute x^n?
- Can we use a divide-and-conquer approach to make it faster?

ExpBySquaring(n, x)

if $n > 1$
  $y = ExpBySquaring(\lfloor n/2 \rfloor, x^2)$
  if $odd(n)$ $y = x * y$
  return y
else return x

Merge Sort

function mergesort($a[1 \ldots n]$)
Input: An array of numbers $a[1 \ldots n]$
Output: A sorted version of this array

if $n > 1$:
  return merge(mergesort($a[1 \ldots \lfloor n/2 \rfloor]$), mergesort($a[\lfloor n/2 \rfloor + 1 \ldots n]$))
else:
  return a

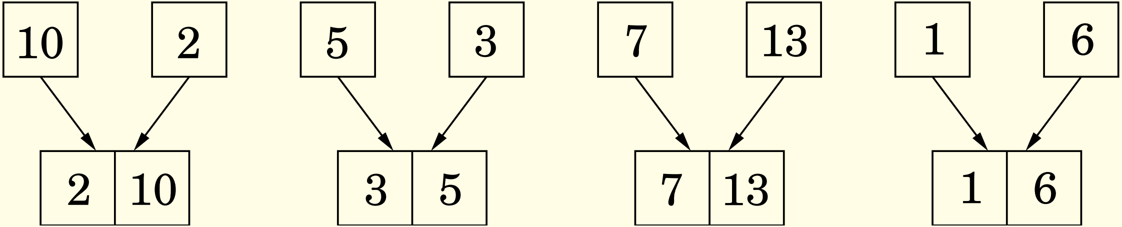
Merge Sort (Continued)

```
function merge(x[1...k], y[1...l])
if k = 0:   return y[1...l]
if l = 0:   return x[1...k]
if x[1] ≤ y[1]:
   return x[1] ∘ merge(x[2...k], y[1...l])
else:
   return y[1] ∘ merge(x[1...k], y[2...l])
```

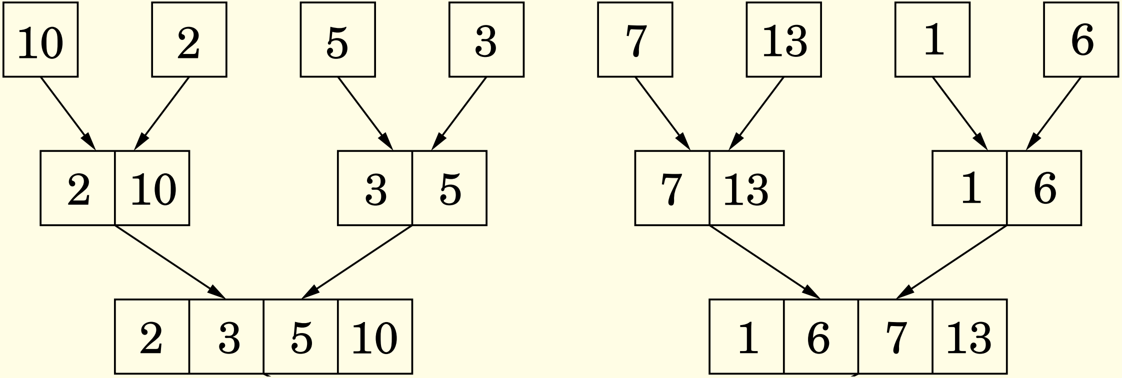
Merge Sort Illustration

$\boxed{10}$ $\boxed{2}$ $\boxed{5}$ $\boxed{3}$ $\boxed{7}$ $\boxed{13}$ $\boxed{1}$ $\boxed{6}$

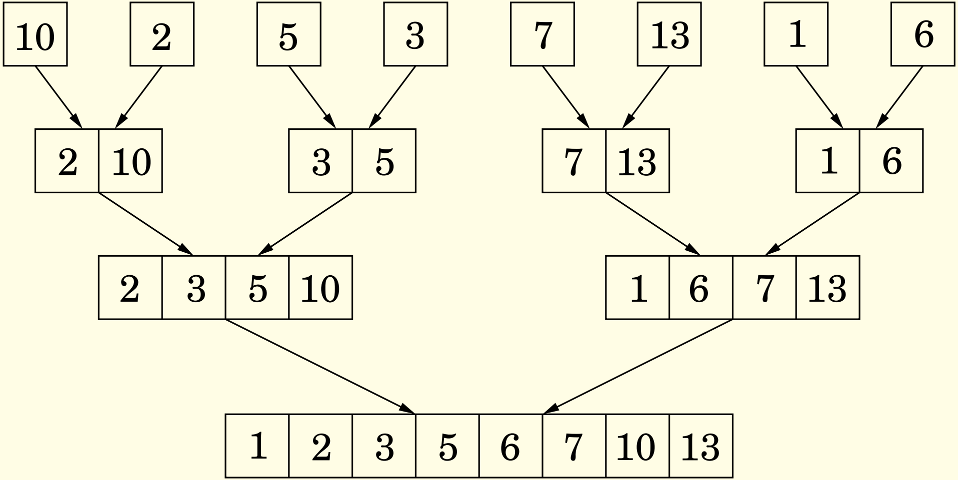
Merge Sort Illustration



Merge Sort Illustration



Merge Sort Illustration



Merge sort time complexity

- $\mathit{mergesort}(A)$ makes two recursive invocations of itself, each with an array half the size of A
- $\mathit{merge}(A, B)$ takes time that is linear in $|A| + |B|$

Merge sort time complexity

- $mergesort(A)$ makes two recursive invocations of itself, each with an array half the size of A
- $merge(A, B)$ takes time that is linear in $|A| + |B|$
- Thus, the runtime is given by the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Merge sort time complexity

- $mergesort(A)$ makes two recursive invocations of itself, each with an array half the size of A
- $merge(A, B)$ takes time that is linear in $|A| + |B|$
- Thus, the runtime is given by the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- In divide-and-conquer algorithms, we often encounter recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

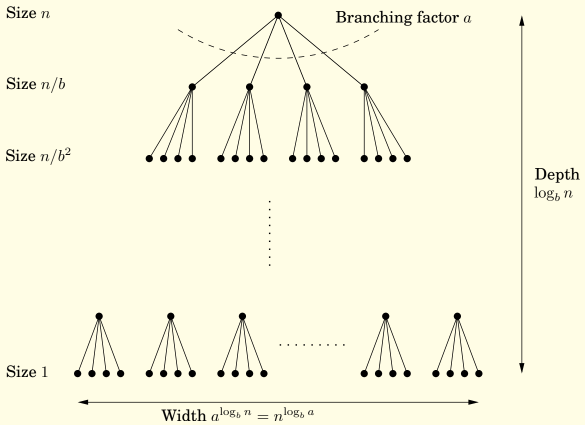
Can we solve them once for all?

Master Theorem

If $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ for constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Proof of Master Theorem



Can be proved by induction, or by summing up the series where each term represents the work done at one level of this tree.

What if Master Theorem can't be appplied?

Look up "Recurrences" from CSE 150

What if Master Theorem can't be appplied?

Look up "Recurrences" from CSE 150

- Guess and check (prove by induction)
  - expand recursion for a few steps to make a guess
  - in principle, can be applied to any recurrence
- Akra-Bazzi method (not covered in class)
  - recurrences can be much more complex than that of Master theorem

Quicksort

$qs(A, l, h)$ /*sorts $A[l \ldots h]$*/

if $l >= h$ **return**;

$(h_1, l_2) =$

$partition(A, l, h)$;

$qs(A, l, h_1)$;

$qs(A, l_2, h)$

Quicksort

$qs(A, l, h)$ /*sorts $A[l \dots h]$*/

if $l >= h$ **return**;
$(h_1, l_2) =$
$partition(A, l, h)$;
$qs(A, l, h_1)$;
$qs(A, l_2, h)$

$partition(A, l, h)$

$k = selectPivot(A, l, h)$; $p = A[k]$;
$swap(A, h, k)$;
$i = l - 1$; $j = h$;
while *true* **do**
    do $i{+}{+}$ **while** $A[i] < p$;
    do $j{-}{-}$ **while** $A[j] > p$;
    if $i \geq j$ **break**;
    $swap(A, i, j)$;
$swap(A, i, h)$
return $(j, i + 1)$

Analysis of Runtime of *qs*

General case: Given by the recurrence $T(n) = n + T(n_1) + T(n_2)$
where n_1 and n_2 are the sizes of the two sub-arrays after partition.

Analysis of Runtime of *qs*

General case: Given by the recurrence $T(n) = n + T(n_1) + T(n_2)$
where n_1 and n_2 are the sizes of the two sub-arrays after partition.

Best case: $n_1 = n_2 = n/2$. By master theorem, $T(n) = O(n \log n)$

Analysis of Runtime of *qs*

General case: Given by the recurrence $T(n) = n + T(n_1) + T(n_2)$
where n_1 and n_2 are the sizes of the two sub-arrays after partition.

Best case: $n_1 = n_2 = n/2$. By master theorem, $T(n) = O(n \log n)$

Worst case: $n_1 = 1$, $n_2 = n - 1$. By master theorem, $T(n) = O(n^2)$

- *A fixed choice of pivot index, say, h, leads to worst-case behavior in common cases, e.g., input is sorted.*

Analysis of Runtime of *qs*

General case: Given by the recurrence $T(n) = n + T(n_1) + T(n_2)$
where n_1 and n_2 are the sizes of the two sub-arrays after partition.

Best case: $n_1 = n_2 = n/2$. By master theorem, $T(n) = O(n \log n)$

Worst case: $n_1 = 1$, $n_2 = n - 1$. By master theorem, $T(n) = O(n^2)$

- *A fixed choice of pivot index, say, h, leads to worst-case behavior in common cases, e.g., input is sorted.*

Lucky/unlucky split: Alternate between best- and worst-case splits.

$$
\begin{aligned}
T(n) &= n + T(1) + \boxed{\text{T(n-1)}} + n \quad \text{(worst case split)} \\
&= n + 1 + \boxed{\text{(n-1) + 2T((n-1)/2)}} = 2n + 2T((n-1)/2)
\end{aligned}
$$

which has an $O(n \log n)$ solution.

Analysis of Runtime of *qs*

General case: Given by the recurrence $T(n) = n + T(n_1) + T(n_2)$ where n_1 and n_2 are the sizes of the two sub-arrays after partition.

Best case: $n_1 = n_2 = n/2$. By master theorem, $T(n) = O(n \log n)$

Worst case: $n_1 = 1$, $n_2 = n - 1$. By master theorem, $T(n) = O(n^2)$

- *A fixed choice of pivot index, say, h, leads to worst-case behavior in common cases, e.g., input is sorted.*

Lucky/unlucky split: Alternate between best- and worst-case splits.

$$
\begin{aligned}
T(n) &= n + T(1) + \boxed{\text{T(n-1)}} + n \quad \text{(worst case split)} \\
&= n + 1 + \boxed{\text{(n-1) + 2T((n-1)/2)}} = 2n + 2T((n-1)/2)
\end{aligned}
$$

which has an $O(n \log n)$ solution.

Three-fourths split:

$$T(n) = n + T(0.25n) + T(0.75n) \leq n + 2T(0.75n) = O(n \log n)$$

Average case analysis of *qs*

Define input distribution: All permutations equally likely

Average case analysis of *qs*

Define input distribution: All permutations equally likely

Simplifying assumption: all elements are distinct. (Nonessential assumption)

Average case analysis of *qs*

Define input distribution: All permutations equally likely

Simplifying assumption: all elements are distinct. (Nonessential assumption)

Set up the recurrence: When all permutations are qually likely, the selected pivot has an equal chance of ending up at the i^{th} position in the sorted order, for all $1 \le i \le n$. Thus, we have the following recurrence for the average case:

$$T(n) = n + \frac{1}{n} \sum_{i=1}^{n-1} (T(i) + T(n-i))$$

Average case analysis of *qs*

Define input distribution: All permutations equally likely

Simplifying assumption: all elements are distinct. (Nonessential assumption)

Set up the recurrence: When all permutations are qually likely, the selected pivot has an equal chance of ending up at the i^{th} position in the sorted order, for all $1 \leq i \leq n$. Thus, we have the following recurrence for the average case:

$$T(n) = n + \frac{1}{n} \sum_{i=1}^{n-1} (T(i) + T(n-i))$$

Solve recurrence: Cannot apply the master theorem, but since it seems that we get an $O(n \log n)$ bound even in seemingly bad cases, we can try to establish a $cn \log n$ bound via induction.

Establishing average case of *qs*

- Establish base case. (Trivial.)
- Induction step involves summation of the form $\sum_{i=1}^{n-1} i \log i$.

  Attempt 1: Bound $\log i$ above by $\log n$. (Induction fails.)

  Attempt 2: Split the sum into two parts:

$$\sum_{i=1}^{n/2} i \log i + \sum_{i=n/2+1}^{n-1} i \log i$$

  and apply the approximation to each half. (Succeeds with $c \geq 4$.)

  Attempt 3: Replace summation with integration. (See "Integration method" in Summations.)

$$\int_{x=1}^{n} x \log x = \frac{x^2}{2}\left(\log x - \frac{1}{2}\right)\Big|_{x=1}^{n}$$

  (Succeeds with the constraint $c \geq 2$.)

Randomized Quicksort

- Picks a pivot at random
- What is its complexity?

Randomized Quicksort

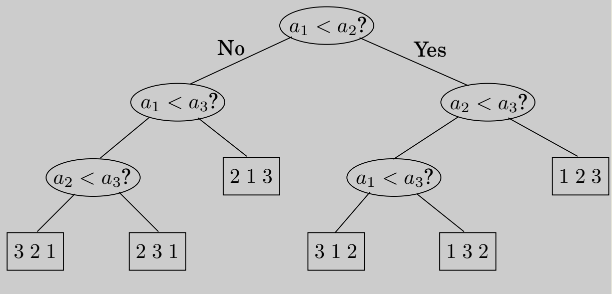
- Picks a pivot at random
- What is its complexity?
  - For randomized algorithms, we talk about *expected complexity*, which is an average over all possible values of the random variable.

Randomized Quicksort

- Picks a pivot at random
- What is its complexity?
  - For randomized algorithms, we talk about *expected complexity*, which is an average over all possible values of the random variable.
- If pivot index is picked uniformly at random over the interval $[l, h]$, then:
  - every array element is equally likely to be selected as the pivot
  - every partition is equally likely
  - thus, *expected* complexity of *randomized* quicksort is given by the same recurrence as the *average* case of qs.

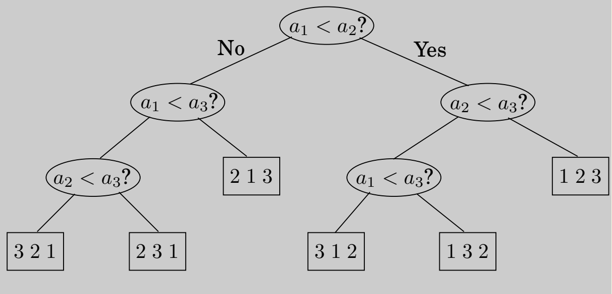
Lower bounds for comparison-based sorting

- Sorting algorithms can be depicted as trees: each leaf identifies the input permutation that yields a sorted order.



Lower bounds for comparison-based sorting

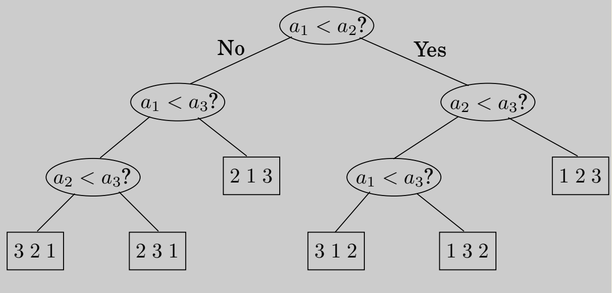
- Sorting algorithms can be depicted as trees: each leaf identifies the input permutation that yields a sorted order.



- The tree has $n!$ leaves, and hence a height of $\log n!$. By Stirling's approximation, $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, so, $\log n! = O(n \log n)$

Lower bounds for comparison-based sorting

- Sorting algorithms can be depicted as trees: each leaf identifies the input permutation that yields a sorted order.



- The tree has $n!$ leaves, and hence a height of $\log n!$. By Stirling's approximation, $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, so, $\log n! = O(n \log n)$
- No *comparison-based* sorting algorithm can do better!

Bucket sort

Overview

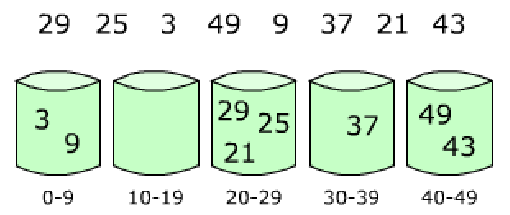
Divide: Partition input into intervals (buckets), based on key values

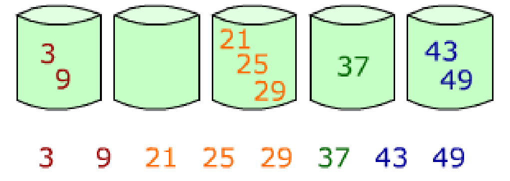
- Linear scan of input, drop into appropriate bucket

Recurse: Sort each bucket

Combine: Concatenate bin contents

Example





Bucket sort (Continued)

- Bucket sort generalizes quicksort to multiple partitions
  - Combination = concatenation
  - Worst case quadratic bound applies
  - But performance can be much better if input distribution is uniform.
    Exercise: What is the runtime in this case?
- Used by letter sorting machines in post offices

Counting Sort

Special case of bucket sort where each bin corresponds to an interval of size 1.

- No need to recurse. Divide $=$ conquered!
- Makes sense only if range of key values is small (usually constant)
- Thus, counting sort can be done in $O(n)$ time!

Counting Sort

Special case of bucket sort where each bin corresponds to an interval of size 1.

- No need to recurse. Divide $=$ conquered!
- Makes sense only if range of key values is small (usually constant)
- Thus, counting sort can be done in $O(n)$ time!
  - *Hmm. How did we beat the $O(n \log n)$ lower bound?*

Radix Sorting

- Treat an integer as a sequence of digits
- Sort digits using counting sort

Radix Sorting

- Treat an integer as a sequence of digits
- Sort digits using counting sort

  LSD sorting: Sort first on least significant digit, and most significant digit last.
  After each round of counting sort, results can be simply concatenated, and given as input to the next stage.

Radix Sorting

- Treat an integer as a sequence of digits
- Sort digits using counting sort

  LSD sorting: Sort first on least significant digit, and most significant digit last.

  After each round of counting sort, results can be simply concatenated, and given as input to the next stage.

  MSD sorting: Sort first on most significant digit, and least significant digit last.

  Unlike LSD sorting, we cannot concatenate after each stage.

Radix Sorting

- Treat an integer as a sequence of digits
- Sort digits using counting sort

  LSD sorting: Sort first on least significant digit, and most significant digit last.

  After each round of counting sort, results can be simply concatenated, and given as input to the next stage.

  MSD sorting: Sort first on most significant digit, and least significant digit last.

  Unlike LSD sorting, we cannot concatenate after each stage.

- *Note:* Radix sort does not divide inputs into smaller subsets

  If you think of input as multi-dimensional data, then we break down the problem to each dimension.

Stable sorting algorithms

- *Stable sorting algorithms:* don't change order of equal elements.
- Merge sort and LSD sort are stable. Quicksort is not stable.

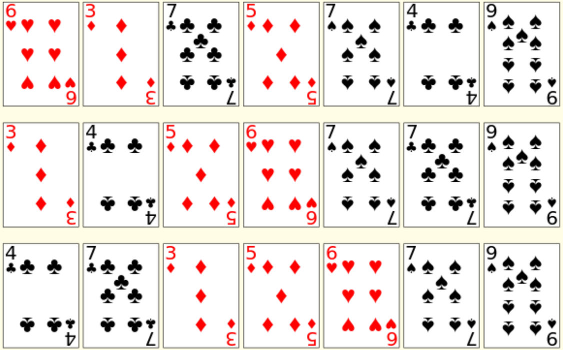
Why is stability important?

Stable sorting algorithms

- *Stable sorting algorithms:* don't change order of equal elements.
- Merge sort and LSD sort are stable. Quicksort is not stable.

Why is stability important?

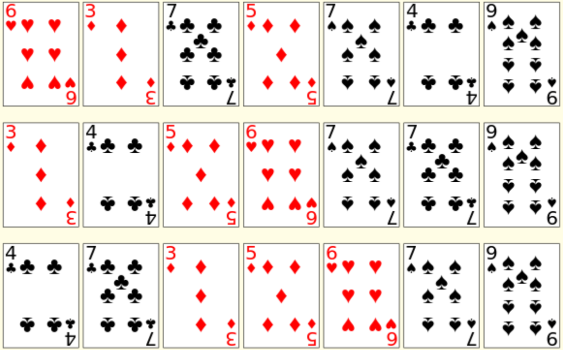
- Effect of sorting on attribute A and then B is the same as sorting on $\langle B, A\rangle$



Images from Wikipedia Commons

Stable sorting algorithms

- *Stable sorting algorithms:* don't change order of equal elements.
- Merge sort and LSD sort are stable. Quicksort is not stable.



Images from Wikipedia Commons

Why is stability important?

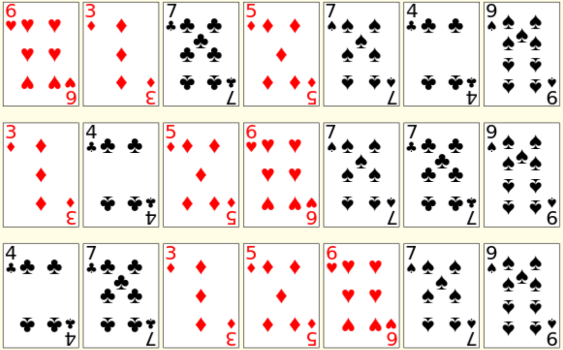
- Effect of sorting on attribute A and then B is the same as sorting on $\langle B, A\rangle$
- LSD sort won't work without this property!

Stable sorting algorithms

- *Stable sorting algorithms:* don't change order of equal elements.
- Merge sort and LSD sort are stable. Quicksort is not stable.

Why is stability important?

- Effect of sorting on attribute A and then B is the same as sorting on $\langle B, A \rangle$
- LSD sort won't work without this property!
- Other examples: sorting spread sheets or tables on web pages



Images from Wikipedia Commons

Sorting strings

- Can use LSD or MSD sorting
  - Easy if all strings are of same length.

Sorting strings

- Can use LSD or MSD sorting
  - Easy if all strings are of same length.
  - Requires a bit more care with variable-length strings.

    Starting point: use a special terminator character $t < a$ for all valid characters a.

Sorting strings

- Can use LSD or MSD sorting
  - Easy if all strings are of same length.
  - Requires a bit more care with variable-length strings.

    Starting point: use a special terminator character $t < a$ for all valid characters a.
- Easy to devise an $O(nl)$ algorithm, where n is the number of strings and l is the maximum size of any string.
  - But such an algorithm is *not* linear in input size.

Sorting strings

- Can use LSD or MSD sorting
  - Easy if all strings are of same length.
  - Requires a bit more care with variable-length strings.
    Starting point: use a special terminator character $t < a$ for all valid characters a.
- Easy to devise an $O(nl)$ algorithm, where n is the number of strings and l is the maximum size of any string.
  - But such an algorithm is *not* linear in input size.
- *Exercise:* Devise a linear-time string algorithm.

  Given a set $\mathcal{S}$ of strings, your algorithm should sort in $O(|\mathcal{S}|)$ time, where

$$|\mathcal{S}| = \sum_{s \in \mathcal{S}} |s|$$

Select k^{th} largest element

Obvious approach: Sort, pick k^{th} element — wasteful, $O(n \log n)$

Better approach: Recursive partitioning, search only on one side

Select k^{th} largest element

Obvious approach: Sort, pick k^{th} element — wasteful, $O(n \log n)$

Better approach: Recursive partitioning, search only on one side

$qsel(A, l, h, k)$

if $l = h$ **return** $A[l]$;
$(h_1, l_2) = partition(A, l, h)$;
if $k \leq h_1$
  return $qsel(A, l, h_1, k)$
else return $qsel(A, l_2, h, k)$

Select k^{th} largest element

Obvious approach: Sort, pick k^{th} element — wasteful, $O(n \log n)$

Better approach: Recursive partitioning, search only on one side

$qsel(A, l, h, k)$

if $l = h$ **return** $A[l]$;
$(h_1, l_2) = partition(A, l, h)$;
if $k \leq h_1$
  return $qsel(A, l, h_1, k)$
else return $qsel(A, l_2, h, k)$

Complexity

Best case: Splits are even: $T(n) = n + T(n/2)$, which has an $O(n)$ solution.

Skewed 10%/90% $T(n) \leq n + T(0.9n)$ — still linear

Worst case: $T(n) = n + T(n-1)$ — *quadratic!*

Worst-case $O(n)$ Selection

Intuition: Spend a bit more time to select a pivot that ensures reasonably balanced partitions

MoM Algorithm [Blum, Floyd, Pratt, Rivest and Tarjan 1973]

Time Bounds for Selection

by .

Manuel Blum, Robert W. Floyd, Vaughan Pratt,
Ronald L. Rivest, and Robert E. Tarjan

Abstract

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm -- PICK. Specifically, no more than 5.4305 n comparisons are ever required. This bound is improved for

$O(n)$ Selection: MoM Algorithm

- Quick select (*qsel*) takes no time to pick a pivot, but then spends $O(n)$ to partition.
- Can we spend more time upfront to make a better selection of the pivot, so that we can avoid highly skewed splits?

Key Idea

- Use the selection algorithm itself to choose the pivot.
  - Divide into sets of 5 elements
  - Compute median of each set ($O(5)$, i.e., constant time)
  - Use selection recursively on these $n/5$ elements to pick their median
    - i.e., choose the median of medians (MoM) as the pivot
- Partition using MoM, and recurse to find kth largest element.

$O(n)$ Selection: MoM Algorithm

Theorem: MoM-based split won't be worse than 30%/70%

Result: Guaranteed linear-time algorithm!

$O(n)$ Selection: MoM Algorithm

Theorem: MoM-based split won't be worse than 30%/70%

Result: Guaranteed linear-time algorithm!

Caveat: The constant factor is non-negligible; use as fall-back if random selection repeatedly yields unbalanced splits.

Selecting maximum element: Priority Queues

Heap

- A tree-based data structure for priority queues

Images from Wikimedia Commons

Selecting maximum element: Priority Queues

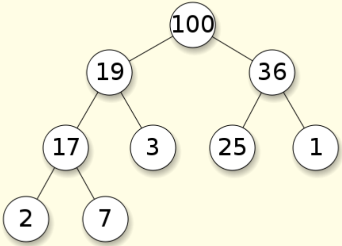
Heap

- A tree-based data structure for priority queues

Heap property: F every subtree h of H

$$\forall k \in keys(h) \quad root(h) \geq k$$

where $keys(h)$ includes all keys within h



Images from Wikimedia Commons

Selecting maximum element: Priority Queues

Heap

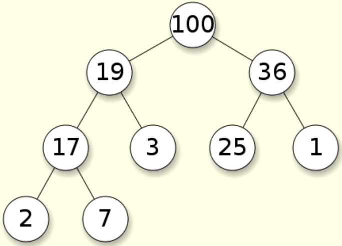
- A tree-based data structure for priority queues

  Heap property: F every subtree h of H

$$\forall k \in keys(h) \quad root(h) \geq k$$

  where $keys(h)$ includes all keys within h

  Note: No ordering of siblings or cousins



Images from Wikimedia Commons

Selecting maximum element: Priority Queues

Heap

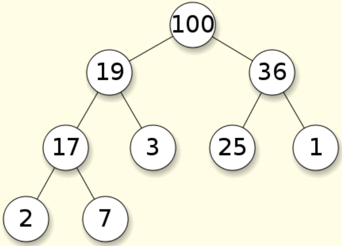
- A tree-based data structure for priority queues

  Heap property: F every subtree h of H

$$\forall k \in keys(h) \quad root(h) \geq k$$

  where $keys(h)$ includes all keys within h

  Note: No ordering of siblings or cousins
- Supports *insert*, *deleteMax* and *max*.



Images from Wikimedia Commons

Selecting maximum element: Priority Queues

Heap

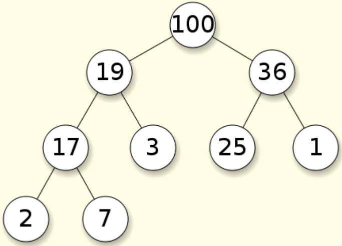
- A tree-based data structure for priority queues

  Heap property: F every subtree h of H

  $$\forall k \in keys(h) \quad root(h) \geq k$$

  where $keys(h)$ includes all keys within h

  Note: No ordering of siblings or cousins
- Supports *insert*, *deleteMax* and *max*.
- Typically implemented using arrays, i.e., without an explicit tree data structure



Images from Wikimedia Commons

Selecting maximum element: Priority Queues

Heap

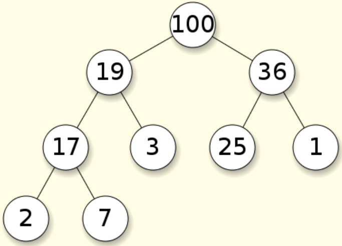
- A tree-based data structure for priority queues

  Heap property: F every subtree h of H

  $$\forall k \in keys(h) \quad root(h) \geq k$$

  where $keys(h)$ includes all keys within h

  Note: No ordering of siblings or cousins
- Supports *insert*, *deleteMax* and *max*.
- Typically implemented using arrays, i.e., without an explicit tree data structure



Task of maintaining max is distributed to subsets of the entire set; alternatively, it can be thought of as maintaining several parallel queues with a single head.

Images from Wikimedia Commons

Binary heap

Array representation: Store heap elements in breadth-first order in the array. Node i's children are at indices $2 * i$ and $2 * i + 1$

- Conceptually, we are dealing with a balanced binary tree

Binary heap

Array representation: Store heap elements in breadth-first order in the array. Node i's children are at indices $2 * i$ and $2 * i + 1$

- Conceptually, we are dealing with a balanced binary tree

Max: Element at the root of the array, extracted in $O(1)$ time

Binary heap

Array representation: Store heap elements in breadth-first order in the array. Node i's children are at indices $2 * i$ and $2 * i + 1$

- Conceptually, we are dealing with a balanced binary tree

Max: Element at the root of the array, extracted in $O(1)$ time

DeleteMax: Overwrite root with last element of heap. Fix heap – takes $O(\log n)$ time, since only the ancestors of the last node need to be fixed up.

Binary heap

Array representation: Store heap elements in breadth-first order in the array. Node i's children are at indices $2 * i$ and $2 * i + 1$

- Conceptually, we are dealing with a balanced binary tree

Max: Element at the root of the array, extracted in $O(1)$ time

DeleteMax: Overwrite root with last element of heap. Fix heap – takes $O(\log n)$ time, since only the ancestors of the last node need to be fixed up.

Insert: Append element to the end of array, fix up heap

Binary heap

Array representation: Store heap elements in breadth-first order in the array. Node i's children are at indices $2 * i$ and $2 * i + 1$

- Conceptually, we are dealing with a balanced binary tree

Max: Element at the root of the array, extracted in $O(1)$ time

DeleteMax: Overwrite root with last element of heap. Fix heap – takes $O(\log n)$ time, since only the ancestors of the last node need to be fixed up.

Insert: Append element to the end of array, fix up heap

MkHeap: Fix up the entire heap. Takes $O(n)$ time.

Binary heap

Array representation: Store heap elements in breadth-first order in the array. Node i's children are at indices $2 * i$ and $2 * i + 1$

- Conceptually, we are dealing with a balanced binary tree

Max: Element at the root of the array, extracted in $O(1)$ time

DeleteMax: Overwrite root with last element of heap. Fix heap – takes $O(\log n)$ time, since only the ancestors of the last node need to be fixed up.

Insert: Append element to the end of array, fix up heap

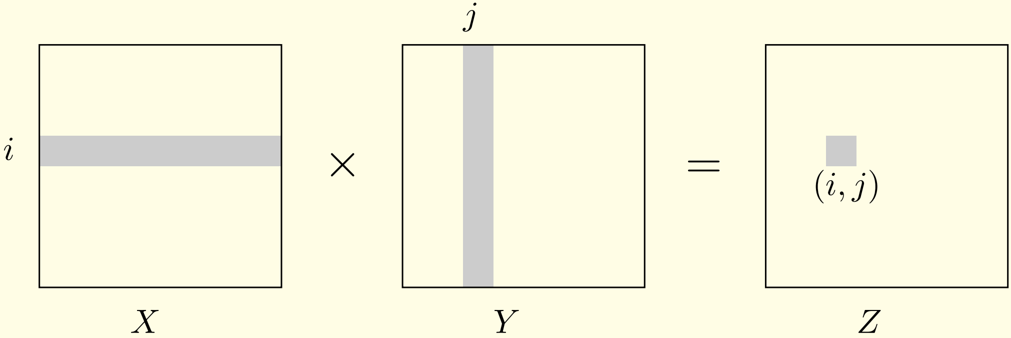
MkHeap: Fix up the entire heap. Takes $O(n)$ time.

Heapsort: $O(n \log n)$ algorithm, *MkHeap* followed by n calls to *DeleteMax*

Matrix Multiplication

The product Z of two $n \times n$ matrices X and Y is given by

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj} \quad \text{— leads to an } O(n^3) \text{ algorithm.}$$



Divide-and-conquer Matrix Multiplication

Divide X and Y into four $n/2 \times n/2$ submatrices

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Recursively invoke matrix multiplication on these submatrices:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Divided, but did not conquer! $T(n) = 8T(n/2) + O(n^2)$, which is still $O(n^3)$

Strassen's Matrix Multiplication

Strassen showed that 7 multiplications are enough:

$$XY = \begin{bmatrix} P_6 + P_5 + P_4 - P_2 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{bmatrix} \quad \text{where}$$

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

Now, the recurrence $T(n) = 7T(n/2) + O(n^2)$ has $O(n^{\log_2 7} = n^{2.81})$ solution!

Best-to-date complexity is about $O(n^{2.4})$, but this algorithm is not very practical.

Karatsuba's Algorithm

Same high-level strategy as Strassen — but predates Strassen.

Divide: n-digit numbers into halves, each with $n/2$-digits:

$$\begin{aligned} a &= \boxed{a_1}\boxed{a_0} = 2^{n/2}a_1 + a_0 \\ b &= \boxed{b_1}\boxed{b_0} = 2^{n/2}b_1 + b_0 \\ ab &= 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + b_1 a_0) + a_0 b_0 \end{aligned}$$

Key point — Instead of 4 multiplications, we can get by with 3 since:

$$a_1 b_0 + b_1 a_0 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0$$

Karatsuba's Algorithm

Same high-level strategy as Strassen — but predates Strassen.

Divide: n-digit numbers into halves, each with $n/2$-digits:

$$\begin{aligned} a &= \boxed{a_1}\boxed{a_0} = 2^{n/2}a_1 + a_0 \\ b &= \boxed{b_1}\boxed{b_0} = 2^{n/2}b_1 + b_0 \\ ab &= 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + b_1 a_0) + a_0 b_0 \end{aligned}$$

Key point — Instead of 4 multiplications, we can get by with 3 since:

$$a_1 b_0 + b_1 a_0 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0$$

Recursively compute $a_1 b_1, a_0 b_0$ and $(a_1 + a_0)(b_1 + b_0)$.

Recurrence $T(n) = 3T(n/2) + O(n)$ has an $O(n^{\log_2 3} = n^{1.59})$ solution!

Karatsuba's Algorithm

Same high-level strategy as Strassen — but predates Strassen.

Divide: n-digit numbers into halves, each with $n/2$-digits:

$$\begin{aligned} a &= \boxed{a_1}\boxed{a_0} = 2^{n/2}a_1 + a_0 \\ b &= \boxed{b_1}\boxed{b_0} = 2^{n/2}b_1 + b_0 \\ ab &= 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + b_1 a_0) + a_0 b_0 \end{aligned}$$

Key point — Instead of 4 multiplications, we can get by with 3 since:

$$a_1 b_0 + b_1 a_0 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0$$

Recursively compute $a_1 b_1$, $a_0 b_0$ and $(a_1 + a_0)(b_1 + b_0)$.

Recurrence $T(n) = 3T(n/2) + O(n)$ has an $O(n^{\log_2 3} = n^{1.59})$ solution!

Note: This trick for using 3 (not 4) multiplications first noted by Gauss (1777-1855).

Faster algorithms for Integer Multiplication

- Toom-Cook Multiplication: Generalize Karatsuba
  - Divide into $n > 2$ parts
- FFT (Fast Fourier Transformation) based multiplication (Schonhage-Strassen)
- Can be more easily understood when integer multiplication is viewed as a polynomial multiplication.

Integer Multiplication Revisited

- An integer represented using digits

$$a_{n-1}\ldots a_0$$

over a base d (i.e., $0 \leq a_i < d$) is very similar to the polynomial

$$A(x) = \sum_{i=0}^{n-1} a_i x^i$$

Specifically, the value of the integer is $A(d)$.

- Integer multiplication follows the same steps as polynomial multiplication:

$$a_{n-1}\ldots a_0 \times b_{n-1}\ldots b_0 = (A(x) \times B(x))(d)$$

Polynomials: Basic Properties

Horner's rule

An n^{th} degree polynomial $\sum_{i=0}^{n} a_i x^i$ can be evaluated in $O(n)$ time:

$$((\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0)$$

Roots and Interpolation

- An n^{th} degree polynomial $A(x)$ has exactly n roots $r_1, \ldots, r_n$. In general, r_i's are complex and need not be distinct.
- It can be represented as a product of sums using these roots:

$$A(x) = \sum_{i=1}^{n} a_i x^i = \prod_{i=0}^{n} (x_i - r_i)$$

- Alternatively, $A(x)$ can be specified uniquely by specifying $n+1$ points (x_i, y_i) on it, i.e., $A(x_i) = y_i$.

Operations on Polynomials

Representation	Add	Mult
Coefficients	$O(n)$	$O(n^2)$
Roots	?	$O(n)$
Points	$O(n)$	$O(n)$

Note: Point representation is the best for computation! But usually, only the coefficients are given

Solution: Convert to point form by *evaluating* $A(x)$ at selected points.

But conversion defeats the purpose: requires $O(n)$ evaluations, each taking $O(n)$ time, thus we are back to $O(n^2)$ total time.

Toom (and FFT) Idea: Choose evaluation points judiciously to speed up evaluation

Integer Multiplication Summary

- Algorithms implemented in libraries for arbitrary precision arithmetic, with applications in public key cryptography, computer algebra systems, etc.
- GNU MP is a popular library, uses various algorithms based on input size: naive, Karatsuba, Toom-3, Toom-4, or Schonhage-Strassen (at about 50K digits).
- Karatsuba is Toom-2. Toom-N is based on
  - Evaluating a polynomial at $2N$ points,
  - performing point-wise multiplication, and
  - interpolating to get back the polynomial, while
  - minimizing the operations needed for interpolation

Integer Multiplication Summary

- Algorithms implemented in libraries for arbitrary precision arithmetic, with applications in public key cryptography, computer algebra systems, etc.
- GNU MP is a popular library, uses various algorithms based on input size: naive, Karatsuba, Toom-3, Toom-4, or Schonhage-Strassen (at about 50K digits).
- Karatsuba is Toom-2. Toom-N is based on
  - Evaluating a polynomial at $2N$ points,
  - performing point-wise multiplication, and
  - interpolating to get back the polynomial, while
  - minimizing the operations needed for interpolation