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

3. Selection
   Select $k$-th min

4. Multiplication
   Matrix
   Multiplication
   Integer
   multiplication

Recurrences
Quicksort
Lower Bound
Radix sort
Priority Queues
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 $\text{root}(T)$, 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 $\text{root}(T)$, 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.
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?

```python
ExpBySquaring(n, x)
    if n > 1
        y = ExpBySquaring(⌊n/2⌋, x^2)
        if odd(n) y = x * y
    return y
else return x
```
Merge Sort

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

if n > 1:
    return merge(mergesort(a[1...[n/2]]), mergesort(a[[n/2] + 1...n]))
else:
    return a
Merge Sort (Continued)

The ultimate divide-and-conquer algorithm is, of course, binary search: to find a key \( k \) in a large file containing keys \( z[0, 1, \ldots, n-1] \) in sorted order, we first compare \( k \) with \( z[n/2] \), and depending on the result we recurse either on the first half of the file, \( z[0, \ldots, n/2-1] \), or on the second half, \( z[n/2, \ldots, n-1] \). The recurrence now is \( T(n) = T(\lceil n/2 \rceil) + O(1) \), which is the case \( a=1, b=2, d=0 \). Plugging into our master theorem we get the familiar solution: a running time of just \( O(\log n) \).

2.3 Mergesort

The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sublists.

function mergesort(a[1..n])
    Input: An array of numbers
    Output: A sorted version of this array
    if \( n > 1 \):
        return merge(mergesort(a[1..\lceil n/2 \rceil]), mergesort(a[\lceil n/2 \rceil+1..n]))
    else:
        return a

The correctness of this algorithm is self-evident, as long as a correct merge subroutine is specified. If we are given two sorted arrays \( x[1..k] \) and \( y[1..l] \), how do we efficiently merge them into a single sorted array \( z[1..k+l] \)? Well, the very first element of \( z \) is either \( x[1] \) or \( y[1] \), whichever is smaller. The rest of \( z[\cdot] \) can then be constructed recursively.

function merge(x[1..k], y[1..l])
    if \( k = 0 \): return y[1..l]
    if \( l = 0 \): return x[1..k]
    if \( x[1] \leq y[1] \):
        return x[1] \circ merge(x[2..k], y[1..l])
    else:
        return y[1] \circ merge(x[1..k], y[2..l])

Here \( \circ \) denotes concatenation. This merge procedure does a constant amount of work per recursive call (provided the required array space is allocated in advance), for a total running time of \( O(k+l) \). Thus merge's are linear, and the overall time taken by mergesort is \( T(n) = 2T(n/2) + O(n) \), or \( O(n \log n) \).

Looking back at the mergesort algorithm, we see that all the real work is done in merging, which doesn't start until the recursion gets down to singleton arrays. The singletons are...
Merge Sort Illustration

Figure 2.4 The sequence of merge operations in mergesort.

Input: 10 2 3 1 13 5 7 6

merged in pairs, to yield arrays with two elements. Then pairs of these 2-tuples are merged, producing 4-tuples, and so on. Figure 2.4 shows an example.

This viewpoint also suggests how mergesort might be made iterative. At any given moment, there is a set of “active” arrays—initially, the singletons—which are merged in pairs to give the next batch of active arrays. These arrays can be organized in a queue, and processed by repeatedly removing two arrays from the front of the queue, merging them, and putting the result at the end of the queue.

In the following pseudocode, the primitive operation inject adds an element to the end of the queue while eject removes and returns the element at the front of the queue.

```
function iterative-mergesort([a_1, a_2, ..., a_n])
  Input: elements a_1, a_2, ..., a_n to be sorted
  Q = [] (empty queue)
  for i = 1 to n:
    inject(Q, [a_i])
  while |Q| > 1:
    inject(Q, merge(eject(Q), eject(Q)))
  return eject(Q)
```
Merge Sort Illustration

2 3 10 1 6 7 13
10 2 5 3 13 7 1
2 5 3 7 13 1 6
1 6 10 13 3 2 5

In the following pseudocode, the primitive operation `inject` adds an element to the end of the queue while `eject` removes and returns the element at the front of the queue.

```plaintext
function iterative-mergesort(a[1..n])
Input: elements a₁, a₂, …, aₙ to be sorted
Q = [] (empty queue)
for i = 1 to n:
ineject(Q, [aᵢ])
while |Q| > 1:
ineject(Q, merge(eject(Q), eject(Q)))
return eject(Q)
```
Merge Sort Illustration

Input: 10, 2, 3, 1, 13, 5, 7, 6

Mergesort

2 3 10 1 6 7 135
10 2 5 3 13 7 1 6
2 5 3 7 13 1 6 10

The sequence of merge operations in mergesort.

Figure 2.4

function iterative-mergesort
(a[1..n])
Input: elements a1, a2, ..., an to be sorted
Q = [] (empty queue)
for i = 1 to n:
inject(Q, [ai])
while |Q| > 1:
inject(Q, merge(eject(Q), eject(Q)))
return eject(Q)
Merge Sort Illustration

Input: 10 2 3 1 13 5 7 6

merged in pairs, to yield arrays with two elements. Then pairs of these 2-tuples are merged, producing 4-tuples, and so on. Figure 2.4 shows an example.

This viewpoint also suggests how mergesort might be made iterative. At any given moment, there is a set of “active” arrays—initially, the singletons—which are merged in pairs to give the next batch of active arrays. These arrays can be organized in a queue, and processed by repeatedly removing two arrays from the front of the queue, merging them, and putting the result at the end of the queue.

In the following pseudocode, the primitive operation \textit{inject} adds an element to the end of the queue while \textit{eject} removes and returns the element at the front of the queue.

\begin{verbatim}
function iterative-mergesort(a[1..n])
  Input: elements $a_1, a_2, \ldots, a_n$ to be sorted
  \textit{Q} = [ ] (empty queue)
  for $i = 1$ to $n$:
    \textit{inject}(\textit{Q}, \{a_i\})
  while $|\textit{Q}| > 1$:
    \textit{inject}(\textit{Q}, \textit{merge}(\textit{eject}(\textit{Q}), \textit{eject}(\textit{Q})))
  return \textit{eject}(\textit{Q})
\end{verbatim}
Merge sort time complexity

- $\text{mergesort}(A)$ makes two recursive invocations of itself, each with an array half the size of $A$
- $\text{merge}(A, B)$ 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 applied?

Look up “Recurrences” from CSE 150
What if Master Theorem can’t be applied?

- 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

Look up “Recurrences” from CSE 150
QuickSort

\[ \text{qs}(A, l, h) \quad /*\text{sorts } A[l \ldots h] */ \]

\[
\text{if } l \geq h \text{ return;}
\]

\[
(h_1, l_2) =
\]

\[
\text{partition}(A, l, h);
\]

\[
\text{qs}(A, l, h_1);
\]

\[
\text{qs}(A, l_2, h)
\]
# Quicksort

- **Quicksort**

```c
qs(A, l, h) /*sorts A[l . . . h]*/

if l >= h return;

(h1, l2) =
    partition(A, l, h);
qs(A, l, h1);
qs(A, l2, h)
```

- **partition(A, l, h)**

```c
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 ≥ 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.

\[
T(n) = n + T(1) + T(n-1) + n \quad \text{(worst case split)}
\]

\[
= n + 1 + (n-1) + 2T((n-1)/2) = 2n + 2T((n - 1)/2)
\]

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.

$$T(n) = n + T(1) + T(n-1) + n \quad \text{(worst case split)}$$

$$= n + 1 + T((n-1)/2) + n = 2n + 2T((n-1)/2)$$

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 equally 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:

\[
T(n) = n + \frac{1}{n} \sum_{i=1}^{n-1} (T(i) + T(n - i))
\]
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 equally 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*.)

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

  (Succeeds with the constraint $c \geq 2$.)
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, the expected complexity of randomized quicksort is given by the same recurrence as the average case of quicksort.
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

```
29 25  3 49  9 37 21 43
0-9 10-19 20-29 30-39 40-49
```

```
3  9
0-9
```

```
29 25  3 49  9 37 21 43
10-19 20-29 30-39 40-49
```

```
3  9  21 25 29 37 43 49
```

```
3  9  21 25 29 37 43 49
```
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.

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
Sorting strings

- Can use LSD or MSD sorting
  - Easy if all strings are of same length.
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 $S$ of strings, your algorithm should sort in $O(|S|)$ time, where $|S| = \sum_{s \in S} |s|$.
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 \( S \) of strings, your algorithm should sort in \( O(|S|) \) time, where
\[
|S| = \sum_{s \in S} |s|
\]
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

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

1. if $l = h$ return $A[l]$
2. $(h_1, l_2) = partition(A, l, h)$
3. if $k \leq h_1$
   - return $qsel(A, l, h_1, k)$
4. else return $qsel(A, l_2, h, k)$
Select $k^{\text{th}}$ largest element

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

Better approach: Recursive partitioning, search only on one side

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

\[
\begin{align*}
\text{if } l &= h \text{ return } A[l]; \\
(h_1, l_2) &= \text{partition}(A, l, h); \\
\text{if } k &\leq h_1 \\
&\text{return } qsel(A, l, h_1, k) \\
\text{else return } qsel(A, l_2, h, k)
\end{align*}
\]

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 $5.4305n$ comparisons are ever required. This bound is improved for
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 $k$th largest element.
**$O(n)$ Selection: MoM Algorithm**

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

*Result:* Guaranteed linear-time 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: For every subtree $h$ of $H$

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

where $\text{keys}(h)$ includes all keys within $h$
Selecting maximum element: Priority Queues

**Heap**

- A tree-based data structure for priority queues

  **Heap property:** For every subtree $h$ of $H$
  
  $$\forall k \in \text{keys}(h) \quad \text{root}(h) \geq k$$

  where $\text{keys}(h)$ 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:** For every subtree $h$ of $H$
  
  $$\forall k \in \text{keys}(h) \quad \text{root}(h) \geq k$$
  
  where $\text{keys}(h)$ includes all keys within $h$

  **Note:** No ordering of siblings or cousins

- Supports `insert`, `deleteMax` and `max`.

Images from [Wikimedia Commons](https://commons.wikimedia.org)
Selecting maximum element: Priority Queues

Heap

- A tree-based data structure for priority queues
- **Heap property:** For every subtree \( h \) of \( H \)
  \[
  \forall k \in \text{keys}(h) \quad \text{root}(h) \geq k
  \]
  where \( \text{keys}(h) \) includes all keys within \( h \)
- **Note:** No ordering of siblings or cousins
- Supports \textit{insert}, \textit{deleteMax} and \textit{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:** For every subtree $h$ of $H$

\[ \forall k \in \text{keys}(h) \quad \text{root}(h) \geq k \]

where \( \text{keys}(h) \) includes all keys within \( h \)
- **Note:** No ordering of siblings or cousins
- Supports \textit{insert}, \textit{deleteMax} and \textit{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 $2 \times i$ and $2 \times i + 1$

- 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 $2 \times i$ and $2 \times i + 1$

- 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 $2 \times i$ and $2 \times i + 1$

- 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 $2 \times i$ and $2 \times i + 1$

- 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 $2 \times i$ and $2 \times i + 1$

- 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 $2 \times i$ and $2 \times i + 1$

- 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

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

— leads to an $O(n^3)$ algorithm.

This follows by taking expected values of both sides of the following statement:

Time taken on an array of size $n \leq (\text{time taken on an array of size } 3n/4) + (\text{time to reduce array size to } \leq 3n/4)$,

and, for the right-hand side, using the familiar property that the expectation of the sum is the sum of the expectations.

From this recurrence we conclude that $T(n) = O(n)$: on any input, our algorithm returns the correct answer after a linear number of steps, on the average.

The Unix `sort` command

Comparing the algorithms for sorting and median-finding we notice that, beyond the common divide-and-conquer philosophy and structure, they are exact opposites. Mergesort splits the array in two in the most convenient way (first half, second half), without any regard to the magnitudes of the elements in each half; but then it works hard to put the sorted subarrays together. In contrast, the median algorithm is careful about its splitting (smaller numbers first, then the larger ones), but its work ends with the recursive call.

Quicksort is a sorting algorithm that splits the array in exactly the same way as the median algorithm; and once the subarrays are sorted, by two recursive calls, there is nothing more to do. Its worst-case performance is $\Theta(n^2)$, like that of median-finding. But it can be proved (Exercise 2.24) that its average case is $O(n \log n)$; furthermore, empirically it outperforms other sorting algorithms. This has made quicksort a favorite in many applications—for instance, it is the basis of the code by which really enormous files are sorted.
Divide-and-conquer Matrix Multiplication

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

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

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:

\[ XY = \begin{bmatrix} P_6 + P_5 + P_4 - P_2 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{bmatrix} \]

where

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

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:

- \[ a = \begin{array}{c} a_1 \\ a_0 \end{array} = 2^{n/2} a_1 + a_0 \]
- \[ b = \begin{array}{c} b_1 \\ b_0 \end{array} = 2^{n/2} b_1 + b_0 \]
- \[ ab = 2^n a_1 b_1 + 2^{n/2} (a_1 b_0 + b_1 a_0) + a_0 b_0 \]

**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:

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

**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:

\[
\begin{align*}
 a &= \begin{array}{c|c} a_1 & a_0 \end{array} = 2^{n/2} a_1 + a_0 \\
 b &= \begin{array}{c|c} b_1 & b_0 \end{array} = 2^{n/2} b_1 + b_0 \\
 ab &= 2^n a_1 b_1 + 2^{n/2} (a_1 b_0 + b_1 a_0) + a_0 b_0
\end{align*}
\]

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 $r_1, ..., r_n$. 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

<table>
<thead>
<tr>
<th>Representation</th>
<th>Add</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coefficients</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Roots</td>
<td>?</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Points</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

**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