Medians and Order Statistics

CSE 5311: Design and Analysis of Algorithms

Alex Dillhoff

The University of Texas at Arlington

Introduction

Order Statistics

Minimum and Maximum

Selection in Expected Linear Time

Selection in Worst-Case Linear Time

Introduction

Introduction

Computing medians and order statistics has many applications in computer science, signal processing, data analysis, and machine learning, for example.

This lecture reviews the basic concepts of medians and order statistics, and presents algorithms for computing them.

Applications

Computer Science

- **Optimizing Quicksort**: the median-of-medians algorithm can be used to select a good pivot.
- **Quickselect**: a selection algorithm that finds the kth smallest element in an unsorted list.

Applications

Image Processing

- **Median Filtering**: a nonlinear digital filtering technique that removes noise from an image.
- **Temporal Noise Reduction:** medians can be used across temporal sequences to reduce noise.

Applications

Data Analysis

- **Percentiles and Quantiles:** medians are a special case of percentiles.
- **Robust Statistics:** medians are more robust to outliers than means.

Applications

Machine Learning

- **Feature Engineering:** medians capture the variability or skewness, which can lead to more robust models.
- **Anomaly Detection:** medians can be used to detect outliers in data.

Order Statistics

Order Statistics

The i^{th} order statistic is the i^{th} smallest element in a set of n elements.

The median is the $\frac{n}{2}^{\text{th}}$ order statistic.

Order Statistics

The minimum and maximum are the 1^{st} and n^{th} order statistics, respectively.

When n is even, there are two medians:

1. the lower median $\frac{n}{2}^{\text{th}}$ and
2. the upper median $\frac{n}{2} + 1^{\text{th}}$.

Order Statistics

The goal for this lecture is to investigate two different approaches for computing order statistics:

1. A linear time algorithm
2. A divide-and-conquer approach

Minimum and Maximum

Minimum and Maximum

What is the lower bound on the number of comparisons needed to find either the maximum or minimum of a set of n elements?

Minimum and Maximum

What is the lower bound on the number of comparisons needed to find either the maximum or minimum of a set of n elements?

n-1 comparisons

Minimum and Maximum

One such argument could be that if we left even 1 comparison out of the $n - 1$ comparisons, we could not guarantee that we had found the minimum or maximum.

When implementing an algorithm, it is reasonable to conclude that an optimal implementation would require $n - 1$ comparisons.

Minimum and Maximum

There are plenty of algorithms that we implement which are not optimal in terms of their theoretical lower bound.

Consider a naive matrix multiplication algorithm: there are many redundant reads from memory in this algorithm.

Minimum and Maximum

For example, if we compute $C = AB$, we need to calculate the output values $C_{1,1}$ and $C_{1,2}$, among others.

Both of these outputs require reading from the first row of A.

Minimum and Maximum

We could find both the minimum and maximum of a set in $2n - 2$ operations by passing over the set twice.

This is theoretically optimal since each pass is performing the optimal $n - 1$ comparisons.

Minimum and Maximum

Can we do this with a single pass?

Minimum and Maximum

Can we do this with a single pass?

- Compare a pair of elements with each other.
- Compare them to the minimum and maximum.
- Total comparisons: $3 \left\lfloor \frac{n}{2} \right\rfloor$ comparisons.

Selection in Expected Linear Time

Selection in Expected Linear Time

We now turn to the problem of **selection**.

Given a set of n elements and an integer i, we want to find the i^{th} order statistic.

Assumptions:

- All elements are distinct.
- i is between 1 and n.

Selection in Expected Linear Time

The randomized select algorithm returns the i^{th} smallest element of an array bounded between indices p and r.

It relies on `randomized_partition`, just like Quicksort.

Randomized Select

```
def randomized_select(A, p, r, i):
    if p == r:
        return A[p]
    q = randomized_partition(A, p, r)
    k = q - p + 1
    if i == k:
        return A[q]
    elif i < k:
        return randomized_select(A, p, q-1, i)
    else:
        return randomized_select(A, q+1, r, i-k)
```

Randomized Select

Example: Find median in $A = [3, 2, 6, 1, 5, 4, 7]$

Analysis

The worst-case running time of `randomized_select` is $O(n^2)$ since we are partitioning n elements at $\Theta(n)$ each.

Since the pivot of `randomized_partition` is selected at random, we can expect a *good* split at least every 2 times it is called.

Analysis

The proof for this is similar to the one made when analyzing Quicksort.

The expected number of times we must partition before we get a helpful split is 2, which only doubles the running time.

The recurrence is still $T(n) = T(3n/4) + \Theta(n) = \Theta(n)$.

Analysis

The first step to showing that the expected runtime of `randomized_select` is $\Theta(n)$ is...

Show that a partitioning is helpful with probability at least $\frac{1}{2}$.

Terms

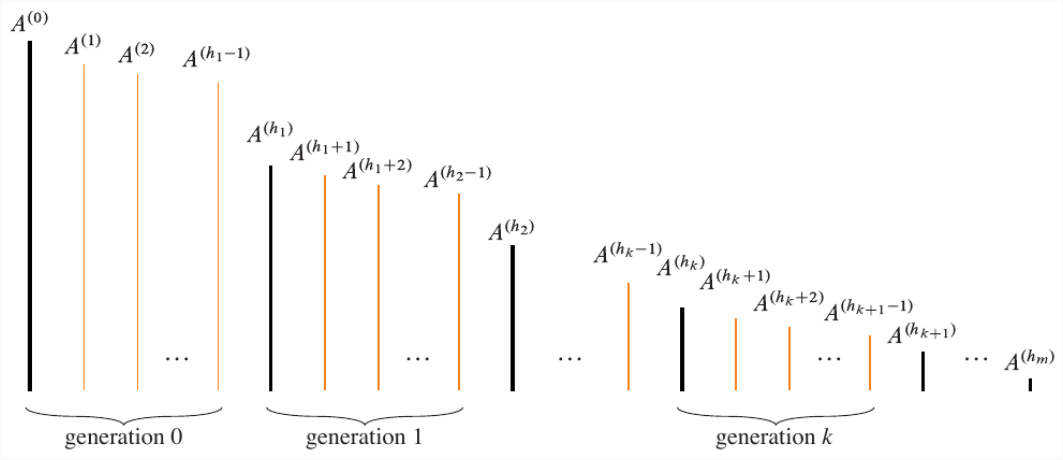
- h_i is the event that the i^{th} partitioning is helpful.
- $\{h_0, h_1, \ldots, h_m\}$ is the sequence of helpful partitionings.
- $n_k = |A^{(h_k)}|$ is the number of elements in the subarray $A^{(h_k)}$ at the k^{th} partitioning.
- $n_k \leq (3/4)n_{k-1}$ for $k \geq 1$, or $n_k \leq (3/4)^k n_0$.
- $X_k = h_{k+1} - h_k$ is the number of unhelpful partitionings between the k^{th} and $(k+1)^{\text{th}}$ helpful partitionings.

Analysis

There are certainly partitionings that are not helpful.

These are depicted as subarrays within each generation of helpful partitionings.

Analysis



The sets within each generation of helpful partitionings are not helpful.

Analysis

Given that the probability that a partitioning is helpful is at least $\frac{1}{2}$, we know that $E[X_k] \leq 2$.

An upper bound on the number of comparisons of partitioning can be derived.

Analysis

The total number of comparisons made when partitioning is less than

$$\begin{aligned}\sum_{k=0}^{m-1}\sum_{j=h_k}^{h_k+X_k-1}|A^{(j)}| &\le \sum_{k=0}^{m-1}\sum_{j=h_k}^{h_k+X_k-1}|A^{(h_k)}| \\ &= \sum_{k=0}^{m-1} X_k|A^{(h_k)}| \\ &\le \sum_{k=0}^{m-1} X_k\left(\frac{3}{4}\right)^k n_0.\end{aligned}$$

The first term on the first line represents the total number of comparisons across all sets.

The first sum loops through the m helpful partitionings, and the inner loop sums the number of comparisons made for each unhelpful partitioning.

$$\sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(j)}|$$

It is bounded by the term on the right.

This is because $|A^{(j)}| \leq |A^{(h_k)}|$ if $A^{(j)}$ is in the k^{th} generation of helpful partitionings. Based on the following:

$n_k \leq (3/4)n_{k-1}$ for $k \geq 1$, or $n_k \leq (3/4)^k n_0$.

Analysis

The second line is derived from the fifth term presented earlier:

$X_k = h_{k+1} - h_k$ is the number of unhelpful partitionings between the k^{th} and $(k+1)^{\text{th}}$ helpful partitionings.

$$\sum_{k=0}^{m-1} X_k |A^{(h_k)}|$$

The third line leverages term 4 again:

$n_k \leq (3/4)n_{k-1}$ for $k \geq 1$, or $n_k \leq (3/4)^k n_0$.

$$\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0.$$

Analysis

The sum is a geometric series, and the total number of comparisons is less than

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0\right] &= n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \mathrm{E}[X_k] \\
&\leq 2n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \\
&< 2n_0 \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\
&= 8n_0.
\end{aligned}$$

Analysis

The last line is the result of a geometric series.

This concludes the proof that `randomized_partition` runs in expected linear time.

Selection in Worst-Case Linear Time

Median of Medians

Finding the median value of a set can be performed in linear time without fully sorting the data.

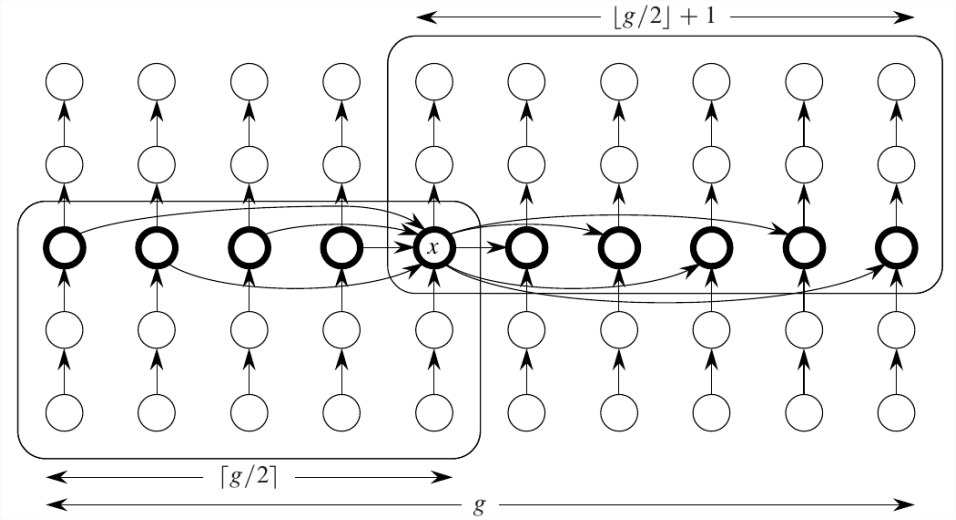
The recurrence is based on discarding a constant fraction of the elements at each step.

We will later show that this algorithm runs in $O(n)$ time.

Median of Medians

1. **Divide**: Partition the set into groups of 5 elements. Depending on the size of the set, there may be less than 5 elements in the last set.
2. **Conquer**: Sort each group and find the median of each group. Since the subsets are of constant size, this is done in constant time.
3. **Combine**: Given the median of each group from step 2, find the median of medians. This value will be used as a pivot for the next step.
4. **Partition**: Use the pivot to separate values smaller and larger than the pivot.
5. **Select**: If the given pivot is the true median based on its position in the original set, select it. If not, recursively select the median from the appropriate partition.

Median of Medians



The median of medians algorithm.

Median of Medians

- Each group is sorted in constant time using an algorithm like insertion sort.
- The median of each group is found in constant time.
- Call `select` recursively on a set of size $n/5$, this returns the pivot x in the figure.
- The pivot is used to partition the set into two sets: $A[1, \ldots, q-1]$ and $A[q+1, \ldots, n]$.

The partitioning call will return the index q of the pivot x.

- Depending on where the pivot q ended up, the relative index k of the i^{th} order statistic needs to be computed.
- If $i = k$, the pivot is the i^{th} order statistic.
- If $i < k$, the i^{th} order statistic is in the left partition.
- If $i > k$, the i^{th} order statistic is in the right partition.

Median of Medians

Example: Median of Medians implementation.

Analysis

The first while loop in the `select` function runs in $O(n)$ time.

Its purpose is to ensure that the input is partitioned into groups of 5 elements.

Analysis

Its purpose is to ensure that the input is partitioned into groups of 5 elements.

- If the input is not divisible by 5, put the smallest element at the beginning of the array.
- If we wanted the smallest element, we are done.
- Otherwise increment p and decrement i so it is no longer in consideration.
- **Each loop runs in $O(n)$ time and will be executed at most 4 times.**

Sorting the groups

Sorting each group is done in constant time since the groups have constant size.

There are $g \leq n/5$ groups, so the total cost is $O(n)$.

First recursive call to `select`

Finding the median of medians via the first recursive call to `select` yields a recurrence of $T(n/5)$.

Partitioning around the pivot

As seen with Quicksort, partitioning around the pivot is done in $O(n)$ time.

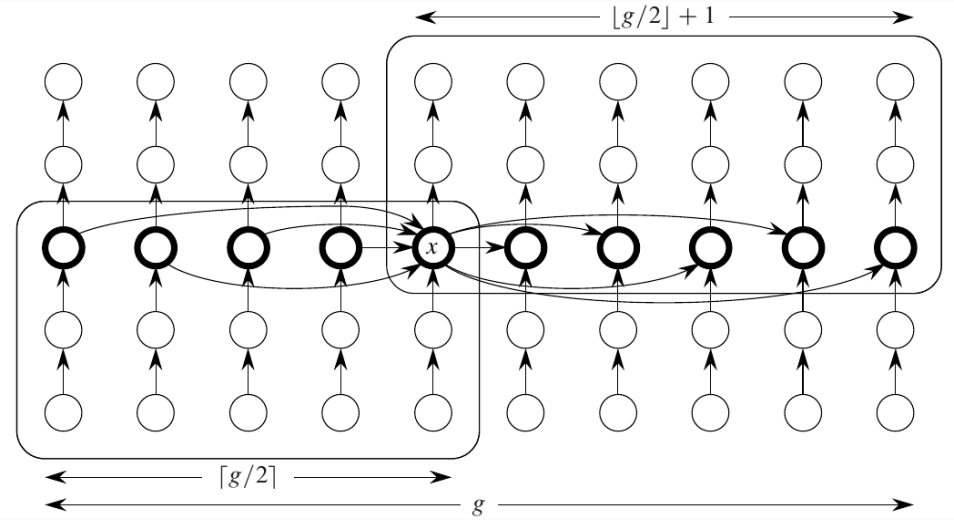
Second recursive call to `select`

One of two other recursive calls to `select` is made.

This call is made on a set of size at most $7n/10$.

To understand why, let's review the figure from before.

Analysis



The median of medians algorithm.

Analysis

The upper-right region contains $\lfloor g/2 \rfloor + 1$ groups.

This means that at least $3(\lfloor g/2 \rfloor + 1) \geq 3g/2$ elements are greater than or equal to the pivot.

Analysis

The lower-left region contains $\lceil g/2 \rceil$ groups.

This means that at least $3\lceil g/2 \rceil$ elements are less than or equal to the pivot.

Analysis

In either case, the recursive call excludes $\geq 3g/2$ elements.

This leaves $5g - 3g/2 = 7g/2 \leq 7n/10$ elements.

Analysis

Adding this up yields the following recurrence:

$$T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$$

Analysis

We now show that $T(n) \leq cn$ for some constant c.

Assume $n \geq 5$.

$$\begin{aligned} T(n) &\leq c(n/5) + c(7n/10) + \Theta(n) \\ &\leq 9cn/10 + \Theta(n) \\ &= cn - cn/10 + \Theta(n) \\ &\leq cn. \end{aligned}$$