Range Queries
(optional: Square Root Complexity)

CENG 213 Data Structures
Yusuf Sahillioğlu
Goal

- Compute a value based on a subarray of an array.
- Consider range [3, 6] below.

\[
\begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 3 & 8 & 4 & 6 & 1 & 3 & 4 \\
\end{array}
\]

- \( \text{sum}_q(3, 6) = 14 \), \( \text{min}_q(3, 6) = 1 \), \( \text{max}_q(3, 6) = 6 \).
Goal

- Compute a value based on a subarray of an array.
- Typical range queries:
  - $\text{sum}_q(a, b)$: calculate the sum of values in range $[a, b]$.
  - $\text{min}_q(a, b)$: find the minimum value in range $[a, b]$.
  - $\text{max}_q(a, b)$: find the maximum value in range $[a, b]$.
Trivial Solution

```c
int sum(int a, int b) {
    int s = 0;
    for (int i = a; i <= b; i++) {
        s += array[i];
    }
    return s;
}
```
Trivial Solution

- Works in $O(n)$ time, where $n$ is the array size.
- We will make this fast!
Static Array Queries

• Assume array is static: values never updated.
• We will handle sum queries and min/max queries in this setting.
Prefix Sum Array

- Value at position $k$ is $\text{sum}_q(0, k)$.
- Can be constructed in $O(n)$ time. How?

Array:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Prefix Sum:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>22</td>
<td>23</td>
<td>27</td>
<td>29</td>
</tr>
</tbody>
</table>
Prefix Sum Array

- Value at position $k$ is $\sum_q(0, k)$.
- Can be constructed in $O(n)$ time. How?
  - Dead simple application of dynamic programming.
    - $P[0]=A[0]$; for ($i=1$ to $n-1$) $P[i]=P[i-1]+A[i]$;

Array:

```
0  1  2  3  4  5  6  7
1  3  4  8  6  1  4  2
```

Prefix Sum:

```
0  1  2  3  4  5  6  7
1  4  8 16 22 23 27 29
```
Prefix Sum Array

• \( \text{sum}_q(a,b) = \text{sum}_q(0,b) - \text{sum}_q(0,a-1) \)
  • Define \( \text{sum}_q(0,-1) = 0 \).

• \( O(n) \): \( \text{sum}_q(3,6) = 8 + 6 + 1 + 4 = 19 \).

• \( O(1) \): \( \text{sum}_q(3,6) = \text{sum}_q(0,6) - \text{sum}_q(0,2) = 27 - 8 \).
Prefix Sum Array

- Can be generalized to higher dimensions.

- Sum of gray subarray: \( S(A) - S(B) - S(C) + S(D) \)
  where \( S(X) \) is the sum of values in a rectangular subarray from the upperleft corner to the position of \( X \).
Sparse Table

• Handles minimum (and similarly max) queries.
• $O(n \log n)$ preprocessing, then all queries in $O(1)$. 
Sparse Table

- Precompute all values of $\min_q(a, b)$ where $b - a + 1$ (the length of the range) is a power of 2.
Sparse Table

• Precompute all values of $\min_q(a, b)$ where $b - a + 1$ (the length of the range) is a power of 2.

• How many precomputed values?
Sparse Table

• Precompute all values of $\min_q(a,b)$ where $b - a + 1$ (the length of the range) is a power of two.

• How many precomputed values?
  • $O(n\log n)$ because
    • there’re $O(\log n)$ range lengths that are powers of 2.
    • there’re $O(n)$ values at each range, e.g., $n$ values for range of length 1, $n-1$ vals for range of length -1, ..
Sparse Table

• Precompute all values of $\min_q(a, b)$ where $b - a + 1$ (the length of the range) is a power of two.

• Each of the $O(n \log n)$ values will be computed in $O(1)$ via the recursion (DP again!):

$$\min_q(a, b) = \min( \min_q(a, a+w-1), \min_q(a+w, b) )$$

where $b-a+1$ is a power of two and $w = (b-a+1)/2 \ //\text{mid.}$
Sparse Table

- Precompute all values of $\min_q(a, b)$ where $b - a + 1$ (the length of the range) is a power of two.
- Each of the $O(n \log n)$ values will be computed in $O(1)$ via the recursion (DP again!):

  $$\ldots$$

- Hence the $O(n \log n)$ preprocessing time.
Sparse Table

- Query response in $O(1)$ via

$$\min_q(a,b) = \min( \min_q(a,a+k-1), \min_q(b-k+1,b) )$$

where $k$ is the largest power of 2 that doesn’t exceed $b-a+1$.

Here, the range $[a,b]$ is represented as the union of the ranges $[a,a+k-1]$ and $[b-k+1,b]$, both of length $k$.

Range length 6, the largest power of 2 that doesn’t exceed 6 is 4, $k=4$. 

$\min_q(1,4)=3$ 

$\min_q(3,6)=1$
Dynamic Array Queries

• Now we will enable updates on array, hence dynamic.

• We will handle sum queries, min/max queries, and update queries in this setting.
Binary Indexed Tree*

- Dynamic variant of a Prefix Sum Array.
  - Handles range sum queries in $O(\log n)$ time. //PSA $O(1)$
  - Handles updating a values in $O(\log n)$ time. //PSA not★
  - Using two BITs make min queries possible.
    - This is more complex than using a Segment Tree (later).

★ PSA can handle this but needs $O(n)$ to rebuild PSA again.

★ BIT aka Fenwick Tree.
Binary Indexed Tree

- Tree is conceptual; we actually maintain an array.
  - Array is 1-indexed to make the implementation easier.
Binary Indexed Tree

- Let $p(k)$ denote the largest pow of 2 that divides $k$. We store a BIT as an array such that
  \[ \text{tree}[k] = \sum_{q}(k - p(k)+1,k) \]

- That is, each position $k$ contains the sum of values in a range of the original array whose length is $p(k)$ and that ends at position $k$.
  - See slides 30-31 for the BIT construction.

- Since $p(6) = 2$, tree[6] contains value of $\sum_{q}(5,6)$. 
Binary Indexed Tree

- \( \text{sum}_q(1,k) \) can be computed in \( O(\log n) \) because a range \([1,k]\) can always be divided into \( O(\log n) \) ranges whose sums are stored in the tree.

Array:

BIT:
Binary Indexed Tree

• $\text{sum}_q(1,k)$ can be computed in $O(\log n)$ because a range $[1,k]$ can always be divided into $O(\log n)$ ranges whose sums are stored in the tree.

Array:

```
1 2 3 4 5 6 7 8
1 3 4 8 6 1 4 2
```

BIT:
Binary Indexed Tree

• $\text{sum}_q(1,7) = \text{sum}_q(1,4) + \text{sum}_q(5,6) + \text{sum}_q(7,7)$
  $= 16 + 7 + 4 = 27.$

Array:

BIT:
Binary Indexed Tree

- $\text{sum}_q(a,b) = \text{sum}_q(1,b) - \text{sum}_q(1,a-1)$ //PSA trick for $a>1$
- $\text{sum}_q(3,6) = \text{sum}_q(1,6) - \text{sum}_q(1,2) = 23 - 4 = 19.$

Array:

```
1 2 3 4 5 6 7 8
1 3 4 8 6 1 4 2
```

BIT:

```
1 2 3 4 5 6 7 8
1 4 4 16 6 7 4 29
```
- After updating a value in the array, several values in the BIT should be updated.
- If the value at position 3 changes, the sums of the following ranges change:
Binary Indexed Tree

- After updating a value in the array, several values in the BIT should be updated.
- Each array element belongs to $O(\log n)$ ranges, hence update cost is $O(\log n)$. 
Binary Indexed Tree

• Implementation made efficient via bit operations.

\[ p(k) = k \& -k \] \(\text{//largest pow of 2 that divides } k.\)

\[ //zeroes all the bits except the last set one.\]
\[ //p(6)=2: 0110 \rightarrow 0010, p(7)=1: 0111 \rightarrow 0001, ..\]

• Computation of \( \text{sum}_q(1,k) \):

\( O(\log n) \) values are accessed and each move to the next position takes \( O(1) \) time.

```c
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += tree[k];
        k -= k & -k;
    }
    return s;
}
```
Binary Indexed Tree

• Implementation made efficient via bit operations.
  \[ p(k) = k \& -k \] //largest pow of 2 that divides \( k \)

• Addition of \( x \) to position \( k \):
  • \( O(\log n) \) values are accessed and each move to the next position takes \( O(1) \) time.

```c
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k & -k;
    }
}
```
Binary Indexed Tree

• Implementation made efficient via bit operations.

\[ p(k) = k \& -k \]  //largest pow of 2 that divides \( k \)

• Initial construction of a BIT is \( O(n\log n) \).
  • Initialize all elements to 0.
  • Fill all range sums (of length \( p(k) \)).
    • Call add() \( n \) times using the input values: add(1..n,A[i]).
Binary Indexed Tree

• Implementation made efficient via bit operations.

\[ p(k) = k \& -k \quad \text{//largest pow of 2 that divides } k \]

• Initial construction of a BIT is \( O(n) \).
  • Construct a PSA in \( O(n) \).
  • Fill all range sums (of length \( p(k) \)).
    • Use PSA lookups in \( O(1) \) time per sum.
Segment Tree

- A more general data structure than BIT.
  - BIT supports sum queries (min queries possible but complicated).
  - ST supports sum, min, max, gcd, xor in $O(\log n)$ time.
  - ST takes more memory and is harder to implement.
Segment Tree

- Tree is conceptual; we actually maintain an array.
  - Array is 0-indexed* to make the implementation easier.
  - Array size is a power of 2 to make the implementation easier.
    - Append extra elements to get this property, if necessary.

* Query ranges are 0-based but the tree array 1-based.
Segment Tree

- Each internal tree node stores a value based on an array range whose size is a power of 2.

Array:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

ST \((\text{sum}_q)\):
Segment Tree

- Any range \([a, b]\) can be divided into \(O(\log n)\) ranges whose values are stored in tree nodes.

**Array:**

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
5 & 8 & 6 & 3 & 2 & 7 & 2 & 6 \\
\end{array}
\]

**ST (sum_q):**

\[\text{sum}_q(2, 7) = 9 + 17\]
Segment Tree

- At most 2 nodes on each level needed $\Rightarrow O(\log n)$
  nodes/ranges needed, so $\sum_q$ complexity is $O(\log n)$.

Array:

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
5 & 8 & 6 & 3 & 2 & 7 & 2 & 6
\end{array}
\]

ST ($\sum_q$):

$\sum_q(2,7) = 9 + 17$
Segment Tree

- After an update, update all nodes whose value depends on the updated value.

Array:

ST \( \text{sum}_q \):
Segment Tree

- Do this by traversing the path from the updated element to root and updating nodes along the path.

Array:

ST \( (\text{sum}_q) \):
Segment Tree

• The path from bottom to top always consists of $O(\log n)$ nodes, so update complexity is $O(\log n)$.

Array:

ST ($\sum_q$):
Segment Tree

• Implementation with an array of $2n$ elements where $n$ is the size of the original array and a power of 2.
Segment Tree

- Tree nodes stored from top to bottom.
  - tree[n] to tree[2n-1], the bottom level, input values.
Segment Tree

- Parent of tree[$k$] is tree[$\lfloor k/2 \rfloor$].
- Children of tree[$k$] is tree[$2k$] and tree[$2k+1$].
Segment Tree

- \( \text{sum}_q(a,b) \) in \( O(\log n) \) because ST has \( O(\log n) \) levels and we move one level higher at each step.

```c
int sum(int a, int b) {
    a += n; b += n; //range initially [a+n,b+n].
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

\[ \text{sum}_q(2,7) = 9 + 17 \]
Segment Tree

- `add()` increases the array value at position \( k \) by \( x \) in \( O(\log n) \) because ST has \( O(\log n) \) levels and we move one level higher at each step.

```cpp
void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k] + tree[2*k+1];
    }
}
```
Segment Tree

- ST can be constructed in $O(n)$. How?
Segment Tree

• ST can be constructed in $O(n)$. How?
  • Calling `add n` times on initially 0 array is not $O(n)$. 

![Diagram of a segment tree](image)

```plaintext
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
39 22 17 13 9 9 8 5 8 6 3 2 7 2 6
```
Segment Tree

- Go from the last intermediate node to the first (root), fill their values by adding their children at indices $2k$ and $2k+1$. Each visited once, hence $O(n)$. 
Segment Tree

- ST can also be used for min queries.
- Divide a range into two parts, compute the answer separately for both parts and then combine answers.
- Already did this for the sum queries.
- Similarly, it handles max, gcd, bit op (xor) queries.
Segment Tree

- ST can also be used for min queries.
- Every tree node contains the smallest value in the corresponding array range.
- Instead of sums, minima are computed.
2D Segment Tree

- Segment Tree of Segment Trees.
- Supports rectangular subarray queries to a 2D array.

Array:

```
  7 6 1 6
  8 7 5 2
  3 9 7 1
  8 5 3 8
```

2D ST (sum_q):

![Diagram of 2D Segment Tree]

```
2D Segment Tree

- Segment Tree of Segment Trees.
- Supports rectangular subarray queries to a 2D array.

Array: 

2D ST (sum_q):

Merge 2 rows (column-wise additions) into a new ST
2D Segment Tree

- Segment Tree of Segment Trees.
- Supports rectangular subarray queries to a 2D array.

Array:

2D ST (sum_q):

Sum for gray region can be obtained from the merged ranges.
2D Segment Tree

- Segment Tree of Segment Trees.
- Supports rectangular subarray queries to a 2D array.

Array:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

2D ST \((\text{sum}_q)\):

Sum for gray region can be obtained from the merged ranges
Lazy Propagation

- An optimization to make range updates faster.
- When there are many updates and updates are done on a range, we can postpone some updates and do those updates only when required.
Lazy Propagation

- $s/z$: sum of values in the range / value of a lazy update.
Lazy Propagation

- ST after increasing the elements in \([a,b]\) by 2.

- When the elements in \([a,b]\) are increased by \(u\), we walk from the root towards the leaves and modify the nodes of the tree as follows.
Lazy Propagation

• ST after increasing the elements in \([a,b]\) by 2.

• If \([x,y]\) partially inside \([a,b]\), we increase the \(s\) value of the node by \(hu\), where \(h\) is the size of the intersection of \([a,b]\) and \([x,y]\), and recur.
Lazy Propagation

- ST after increasing the elements in \([a, b]\) by 2.

- If \([x, y]\) completely inside \([a, b]\), we increase the \(z\) value of the node by \(u\), and stop.
Lazy Propagation

- ST after increasing the elements in $[a,b]$ by 2.

- The idea is that updates will be propagated downwards only when it is necessary, which guarantees that the operations are always efficient.
Lazy Propagation

- ST after computing $\text{sum}_q(a, b)$.

- Notice how the lazy update is applied to 28, and propagated below to 8 and 2 (blue part).
Additional Technique

- Increasing the elements in \([a, b]\) by \(x\) can also be done via Difference Array – has nothing to do w/ ST.

**Array:**

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
3 & 3 & 1 & 1 & 1 & 5 & 2 & 2 \\
\end{array}
\]

**DA:**

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
3 & 0 & -2 & 0 & 0 & 4 & -3 & 0 \\
\end{array}
\]

- DA indicates the differences between consecutive values in the original array \(A\).
- Thus, \(A\) is the prefix sum array of the DA.
We can update a range in A by changing just two elements in DA: to increase A\([1,4]\) by 5, it suffices to increase DA\([1]\) by 5 and decrease DA\([5]\) by 5.

General, \([a,b]\) by \(x \rightarrow DA[a]++x\) and \(DA[b+1]--x\), hence just 2 updates to update \(O(n)\)-range: \(O(1)\).
Segment Tree w/ DS Nodes

- Nodes contain data structures that maintain info about the corresponding ranges.
- ST supporting “how many times does $x$ appear in the range $[a,b]$?”.

Array:

```
 3 1 2 3 1 1 1 1 2
```

ST:
Segment Tree w/ DS Nodes

- Nodes contain data structures that maintain info about the corresponding ranges.
- ST supporting “how many times does x appear in the range \([a,b]\)?”.

**Array:**

- **ST:**

  - Query answered by combining results from nodes that belong to the range.
Segment Tree w/ DS Nodes

- Nodes contain data structures that maintain info about the corresponding ranges.
- ST supporting “how many times does $x$ appear in the range $[a,b]$?”.

Array:

```
3 1 2 3 1 1 1 2
```

ST:

```
1 2 3
4 2 2
1 1 2
```

- Answering takes $O(f(n)\log n)$, where $f(n)$ is the time needed for processing a single node during an operation. Linear search above.
Square Root Complexity

- Algorithm w/ a $O(\sqrt{n})$ time complexity.
  - Poor man’s logarithm.
### Square Root Complexity

- A familiar problem: $\sum_q(a,b)$ and update/add.

<table>
<thead>
<tr>
<th></th>
<th>PSA</th>
<th>BIT</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Complexity</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

- Let’s do it this way: $O(\sqrt{n})$ | $O(1)$
Square Root Complexity

• Divide the array into blocks of size $\sqrt{n}$ so that each block contains the sum of elements inside it.

<table>
<thead>
<tr>
<th>21</th>
<th>17</th>
<th>20</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
**Square Root Complexity**

- Update the sum of a *single* block after each update, hence $O(1)$.

<table>
<thead>
<tr>
<th>21</th>
<th>15</th>
<th>20</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Square Root Complexity

- For sum, divide the range into 3 parts s.t. the sum consists of values of single elements (3+6+2) and sums of blocks between them (15+20).

<table>
<thead>
<tr>
<th>21</th>
<th>15</th>
<th>20</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

- # of single elements is $O(\sqrt{n})$ //block size is $\sqrt{n}$.
- # of blocks is $O(\sqrt{n})$ //need $\sqrt{n}$ blocks to save $n$ vals.
- Hence, range sum in $O(\sqrt{n})$ time.
Square Root Complexity

• The purpose of the block size $\sqrt{n}$ is that it balances two things: the array is divided into $\sqrt{n}$ blocks, each of which contains $\sqrt{n}$ elements.

• In practice, divide into $k$ blocks each of which contains $n/k$ elements.

• Optimal parameter depends on the problem & input.
  • If an algo often goes through the blocks but rarely inspects single elements inside the blocks, it may be a good idea to increase block sizes: divide the array into $k < \sqrt{n}$ blocks, each of which contains $n/k > \sqrt{n}$ elements.
Optional Part

- Remaining slides are optional for CENG 213 purposes.
- We dig more into square root complexity with examples from number theory.
- We also present a binary search algorithm for square root computation.
Square Root Complexity

- Some basic things related to prime numbers*
  - Prime or not?
  - Euler’s totient function.

* Prime number: natural number greater than 1 that has no divisors other than 1 and itself: 2, 3, 5, 7, ..
Square Root Complexity

• Is $n$ prime?
Square Root Complexity

• Is \( n \) prime? iterate through all numbers from 2 to \( n-1 \). Return false if division successful. \( O(n) \).
Square Root Complexity

- Is $n$ prime? Iterate through all numbers from 2 to $\sqrt{n}$. Return false if division successful. $O(\sqrt{n})$.

- If a number has a factor larger than $\sqrt{n}$, then it surely has a factor less than $\sqrt{n}$ (already checked); o/w their multiplication would be $>n$, contradiction.

\[
\begin{align*}
36 \\
2 \times 18 & \quad 3 \times 12 & \quad 4 \times 9 & \quad 6 \times 6
\end{align*}
\]
Square Root Complexity

• Is $n$ prime? iterate through all numbers from 2 to $\sqrt{n}$. Return false if division successful. $O(\sqrt{n})$.

• A larger-than-$\sqrt{n}$ factor of $n$ must be multiplied by a smaller factor that has already been checked.

\[
\begin{align*}
36 \\
2 \times 18 \\
3 \times 12 \\
4 \times 9 \\
6 \times 6
\end{align*}
\]
Square Root Complexity

- Is $n$ prime? So, we will go up to $\sqrt{n}$. But 6 by 6 instead of 1 by 1. Still $O(\sqrt{n})$ but cool.
- All primes ($>3$) are of the form $6k\pm1$. Why?
Square Root Complexity

- Is $n$ prime? So, we will go up to $\sqrt{n}$. But 6 by 6 instead of 1 by 1. Still $O(\sqrt{n})$ but cool.
- All primes (>3) are of the form $6k\pm1$ ‘cos all numbers are of the form $6k+i$ for $i=0..5$.
- $6k+0$, $6k+2$, $6k+4$ are even (not prime). $6k+3$ divisible by 3 (not prime).
- So, $6k+1$ and $6k+5$ can be prime. Write as: $6k\pm1$.
- With this in mind, write the primality test code with increments of 6. *see slide 88 for another cool pattern.*
bool isPrime(int n) {
    if (n <= 1) return false; if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;
    for (i = 5; i * i <= n; i += 6)
        if (n % i == 0 || n % (n + 2) == 0) return false;
    return true; // 6k-1  6k+1
Square Root Complexity

- Prime factorization: every number can be broken down into prime factors, i.e., prime numbers are the basic building blocks of all numbers: $12 = 2 \times 2 \times 3$. 
Square Root Complexity

- Prime factorization of $n$ requires a search for prime factors in the range $[2, \sqrt{n}]$, hence $O(\sqrt{n})^*$. 
- There may be at most 1 prime factor in the range $[\sqrt{n}, n]$ ‘cos o/w 2 factors’ multiplication would be $> n$, contradiction.

* We can find the unique prime factors in $O(\sqrt{n})$ by this search but cannot decide their multiplicity. That’s why prime factorization is very slow to solve for big numbers – foundation of cryptography.
Square Root Complexity

- A simple prime factorization algo is Trial Division.

At least 2x more efficient:

```python
1 def trial_division(n):
2     a = []
3     #Prepare an empty list.
4     f = 2
5     #The first possible factor.
6     while n > 1:
7         #While n still has remaining factors...
8             if (n % f == 0):
9                 #The remainder of n divided by f might
10                    a.append(f)
11                     #If so, it divides n. Add f to the
12                     n /= f
13                     #Divide that factor out of n.
14             else:
15                     #But if f is not a factor of n,
16                     f += 1
17                     #Add one to f and try again.
18     return a
19     #Prime factors may be repeated: 12 factors
```

Some prime factorizations:

<table>
<thead>
<tr>
<th>n</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>149</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>150</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>168</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>169</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>170</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>178</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>188</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>189</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>190</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Square Root Complexity

- For a base-2 $m$-digit number $n$, if we go from 3 to only $\sqrt{n}$, $\pi(2^{m/2}) \approx 2^{m/2} / ((m/2)\ln2)$ divisions are required.

$\pi(n)$: prime counting function, \# of primes less than $n$. 

```python
1 def trial_division(n):
2     a = []
3     while n%2 == 0:
4         a.append(2)
5         n/=2
6     f=3
7     while f*f <= n:
8         if (n % f == 0):
10            a.append(f)
11            n /= f
12     else:
13         f += 2
14     if n>1: a.append(n)
15     #Only odd number is possible
16     return a
```
Square Root Complexity

- $\pi(2^{m/2})$ is exponential in $m$, the problem size.
  - Problem size is not $n$ as we’re dealing with 1 number whose value is $n$. 
Square Root Complexity

- Euler’s totient function $\phi(n)$: # of +ve integers less than $n$ that are relatively prime to $n$.

- $\phi(n) = n-1$ if $n$ is prime (top line). Makes sense!
Square Root Complexity

- Euler’s totient function \( \phi(n) \): # of +ve integers less than \( n \) that are relatively prime to \( n \).
- App: a regular \( n \)-gon can be constructed w/ ruler-and-compass technique if \( \phi(n) \) is a power of 2.

- 6-gon creation: ________________________

http://ceng.metu.edu.tr/~vs/rulercompasshexagon-wiki.gif
Square Root Complexity

- Euler’s totient function $\phi(n)$: # of +ve integers less than $n$ that are relatively prime to $n$.
- We don’t need the proper prime factorization since the exponents $\alpha_i$ are not required in $\phi(n)$.

\[ n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k} \]

\[ \phi(n) = n \left( 1 - \frac{1}{p_1} \right) \left( 1 - \frac{1}{p_2} \right) \ldots \left( 1 - \frac{1}{p_k} \right) \]

- Hence, $O(\sqrt{n})$ time required (slide 81).
Square Root Complexity

- A cool pattern for primes: square of a prime is always one more than a multiple of 24.

\[ p^2 = 24k + 1 \]

\[
\begin{align*}
(p-1)(p+1) &= 24k \\
\frac{2m}{2m} &\text{ or } \frac{4n}{4n} / \text{mod} 4, 8, 12, 16, 20, 24 \Rightarrow (p-1)(p+1) \text{ is } 8 \text{ /mod of 8}
\end{align*}
\]
Square Root Computation

- $\sqrt{n}$ computation algorithm in $O(\log n + p)$, where $p$ is the number of digits in the fractional part: $\sqrt{10} = 3.162$ if $p=3$. 


Square Root Computation

• \( \sqrt{n} \) computation algorithm in \( O(\log n + p) \).

• Integer part is found via binary search \((n=10)\):

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
s & m & e \\
\end{array}
\]

\( 5^2 > 10 \) so go to left. \( //e=m-1. \)

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
s & m & e \\
\end{array}
\]

\( 2^2 < 10 \) so go to right. \( //s=m+1. \)

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
s & e \\
\end{array}
\]

\( 3^2 < 10 \) so go to right. \( //s=m+1. \)
Square Root Computation

- $\sqrt{n}$ computation algorithm in $O(\log n + p)$.
- Integer part is found via binary search ($n=10$):

```
1 2 3 4 5 6 7 8 9 10
```

$4^2 > 10$ so go to left. // $e=m-1$.

```
1 2 3 4 5 6 7 8 9 10
```

Break ‘cos $e<s$.

3 vs. 4, 3 wins ‘cos $4^2 > 10$ and no recovery then.

- $O(\log n)$ time for the integer part.
Square Root Computation

- $\sqrt{n}$ computation algorithm in $O(\log n + p)$.
- Fractional part is found via linear search ($p=3$):

  3.?? = 10
  3.1² < 10
  3.2² > 10 //stop,
  keep 1.
Square Root Computation

• \( \sqrt{n} \) computation algorithm in \( O(\log n + p) \).

• Fractional part is found via linear search (\( p=2 \)):

\[
3.?? = 10 \\
3.1^2 < 10 \quad 3.11^2 < 10 \\
3.2^2 > 10 \ //\text{stop, keep 1.} \\
3.12^2 < 10 \\
3.15^2 < 10 \\
3.16^2 < 10 \\
3.17^2 > 10 \ //\text{stop, keep 6.}
\]
Square Root Computation

- \( \sqrt{n} \) computation algorithm in \( O(\log n + p) \).

- Fractional part is found via linear search (\( p=3 \)):

- At most 9 checks for each of \( p \) digits: \( O(p) \).

- Overall, \( O(\log n + p) \approx O(\log n) \) as \( p \) insignificant.