Data Structures

Arnar Bjarni Arnarson

September 30, 2024

School of Computer Science
Reykjavík University

Today's material

- Prerequisites
- Sliding Window
- Heap
- Union-Find
- Precomputations like prefix sums
- Square root decomposition
- Segment trees
- Sparse tables

Prerequisites

We assume you know how to implement the following data structures using only fixed size arrays and pointers/objects:

- Dynamically sized arrays (like vector in C++)
- Singly/doubly linked lists (like list in C++)
- Queue and stack using either of the above

We also assume you have experience using `(unordered_){map,set}`

Sliding Window

A Sum Problem

Problem description
Write a program that, given an integer array of size N, finds the contiguous subarray of size K with the highest sum.

Input description
Input consist of two lines. The first line contains two space separated integers N, the size of the array, where $1 \leq N \leq 10^6$, and K, the size of the subarrays to consider, where $1 \leq K \leq N$. Then second line contains N space separated integers, the values of the array. Each value in the array is between -10^9 and 10^9.

Output description
Output one line, the sum of the highest valued contiguous subarray of size K.

A Sum Problem

Sample input	Sample output
10 4 17 20 0 1 5 24 8 2 4 1	39

Straightforward Solution

```python
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max((highest, total))
print(highest)
```

Straightforward Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max((highest, total))
print(highest)
```

- This solution constructs all size K contiguous subarrays.

Straightforward Solution

```python
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max((highest, total))
print(highest)
```

- This solution constructs all size K contiguous subarrays.
- What is the time complexity?

Straightforward Solution

```python
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max((highest, total))
print(highest)
```

- This solution constructs all size K contiguous subarrays.
- What is the time complexity?
- There are N starting points, each construction takes K steps, so $\mathcal{O}(NK)$.

Straightforward Solution

```python
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max((highest, total))
print(highest)
```

- This solution constructs all size K contiguous subarrays.
- What is the time complexity?
- There are N starting points, each construction takes K steps, so $\mathcal{O}(NK)$.
- Too slow!

Wasted Operations

- The subarray starting at index i has the sum $a_i + a_{i+1} + \cdots + a_{i+k-1}$.

Wasted Operations

- The subarray starting at index i has the sum $a_i + a_{i+1} + \cdots + a_{i+k-1}$.
- The subarray starting at index $i+1$ has the sum $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$.

Wasted Operations

- The subarray starting at index i has the sum $a_i + a_{i+1} + \cdots + a_{i+k-1}$.
- The subarray starting at index $i+1$ has the sum $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$.
- We iterate over the indices $i+1, i+2, \ldots, i+k-1$ twice.

Wasted Operations

- The subarray starting at index i has the sum $a_i + a_{i+1} + \cdots + a_{i+k-1}$.
- The subarray starting at index $i+1$ has the sum $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$.
- We iterate over the indices $i+1, i+2, \ldots, i+k-1$ twice.
- What changes between starting at i vs. starting at $i+1$?

Wasted Operations

- The subarray starting at index i has the sum $a_i + a_{i+1} + \cdots + a_{i+k-1}$.
- The subarray starting at index $i+1$ has the sum $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$.
- We iterate over the indices $i+1, i+2, \ldots, i+k-1$ twice.
- What changes between starting at i vs. starting at $i+1$?
- We subtract a_i.

Wasted Operations

- The subarray starting at index i has the sum $a_i + a_{i+1} + \cdots + a_{i+k-1}$.
- The subarray starting at index $i+1$ has the sum $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$.
- We iterate over the indices $i+1, i+2, \ldots, i+k-1$ twice.
- What changes between starting at i vs. starting at $i+1$?
- We subtract a_i.
- We add a_{i+k}.

Wasted Operations

- The subarray starting at index i has the sum $a_i + a_{i+1} + \cdots + a_{i+k-1}$.
- The subarray starting at index $i+1$ has the sum $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$.
- We iterate over the indices $i+1, i+2, \ldots, i+k-1$ twice.
- What changes between starting at i vs. starting at $i+1$?
- We subtract a_i.
- We add a_{i+k}.
- A shift from the subarray starting at i to the subarray starting at $i+1$ takes $\mathcal{O}(1)$ time.

Wasted Operations

- The subarray starting at index i has the sum $a_i + a_{i+1} + \cdots + a_{i+k-1}$.
- The subarray starting at index $i+1$ has the sum $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$.
- We iterate over the indices $i+1, i+2, \ldots, i+k-1$ twice.
- What changes between starting at i vs. starting at $i+1$?
- We subtract a_i.
- We add a_{i+k}.
- A shift from the subarray starting at i to the subarray starting at $i+1$ takes $\mathcal{O}(1)$ time.
- This is known as the sliding window technique, in this case with a fixed window size.

Sliding Window Solution

```python
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?

Sliding Window Solution

```python
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?
- This solution constructs the first size K contiguous subarray.

Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?
- This solution constructs the first size K contiguous subarray.
- Then, $N - K$ times, an element is removed and another added.

Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?
- This solution constructs the first size K contiguous subarray.
- Then, $N - K$ times, an element is removed and another added.
- Subtracting and adding numbers is constant time so $\mathcal{O}(N)$.

Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?
- This solution constructs the first size K contiguous subarray.
- Then, $N - K$ times, an element is removed and another added.
- Subtracting and adding numbers is constant time so $\mathcal{O}(N)$.
- Fast enough!

A Substring Problem

Problem description
Write a program that, give a string of size N, finds the longest substring with K distinct elements.

Input description
Input consist of two lines. The first line contains two space separated integers N, the size of the string, where $1 \leq N \leq 10^6$, and K, the number of distinct elements the substring must have, where $1 \leq K \leq 26$. Then second line contains a string of length N consisting of English lowercase characters.

Output description
Output one line, the longest substring with K distinct elements. If no such string exists, output "`DOES NOT EXIST`", without quotations.

A Substring Problem

Sample input	Sample output
`14 3` bacdcbcbcbabdb	cdcbcbcb

General Framework

```python
from string import ascii_lowercase
n, k = map(int, input().split())
s = input()

best_ind, best_len = distinct_k(n, k, s)

if best_len == -1:
  print("DOES NOT EXIST")
else:
  print(s[best_ind:best_ind + best_len])
```

Straightforward Solution

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      distinct = 0
      for symbol in ascii_lowercase:
        if symbol in substring:
          distinct += 1
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

Straightforward Solution

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      distinct = 0
      for symbol in ascii_lowercase:
        if symbol in substring:
          distinct += 1
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- What is the time complexity?

Straightforward Solution

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      distinct = 0
      for symbol in ascii_lowercase:
        if symbol in substring:
          distinct += 1
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- What is the time complexity?
- There are $\mathcal{O}(N^2)$ substrings of the string

Straightforward Solution

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      distinct = 0
      for symbol in ascii_lowercase:
        if symbol in substring:
          distinct += 1
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- What is the time complexity?
- There are $\mathcal{O}(N^2)$ substrings of the string
- Checking each one takes us $\mathcal{O}(N)$ time, so $\mathcal{O}(N^3)$ in total.

Straightforward Solution

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      distinct = 0
      for symbol in ascii_lowercase:
        if symbol in substring:
          distinct += 1
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- What is the time complexity?
- There are $\mathcal{O}(N^2)$ substrings of the string
- Checking each one takes us $\mathcal{O}(N)$ time, so $\mathcal{O}(N^3)$ in total.
- Way too slow!

Constant optimization

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      present = [False for _ in range(26)]
      for symbol in substring:
        present[ord(symbol) - ord('a')] = True
      distinct = sum(present)
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

Constant optimization

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      present = [False for _ in range(26)]
      for symbol in substring:
        present[ord(symbol) - ord('a')] = True
      distinct = sum(present)
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- This is a little faster, by a factor of 26 approximately.

Constant optimization

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      present = [False for _ in range(26)]
      for symbol in substring:
        present[ord(symbol) - ord('a')] = True
      distinct = sum(present)
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- This is a little faster, by a factor of 26 approximately.
- Time complexity is the same.

Constant optimization

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      present = [False for _ in range(26)]
      for symbol in substring:
        present[ord(symbol) - ord('a')] = True
      distinct = sum(present)
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- This is a little faster, by a factor of 26 approximately.
- Time complexity is the same.
- Note that `present` barely differs between adjacent values of `end`

Constant optimization

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    for end in range(start, n+1):
      substring = s[start:end]
      present = [False for _ in range(26)]
      for symbol in substring:
        present[ord(symbol) - ord('a')] = True
      distinct = sum(present)
      cur_len = len(substring)
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- This is a little faster, by a factor of 26 approximately.
- Time complexity is the same.
- Note that `present` barely differs between adjacent values of `end`
- Build it as the substring grows.

Incremental

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    present = [False for _ in range(26)]
    for end in range(start, n):
      present[ord(s[end]) - ord('a')] = True
      distinct = sum(present)
      cur_len = end - start + 1
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

Incremental

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    present = [False for _ in range(26)]
    for end in range(start, n):
      present[ord(s[end]) - ord('a')] = True
      distinct = sum(present)
      cur_len = end - start + 1
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- Now each substring is processed in constant time.

Incremental

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    present = [False for _ in range(26)]
    for end in range(start, n):
      present[ord(s[end]) - ord('a')] = True
      distinct = sum(present)
      cur_len = end - start + 1
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- Now each substring is processed in constant time.
- Time complexity is $\mathcal{O}(N^2)$

Incremental

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    present = [False for _ in range(26)]
    for end in range(start, n):
      present[ord(s[end]) - ord('a')] = True
      distinct = sum(present)
      cur_len = end - start + 1
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- Now each substring is processed in constant time.
- Time complexity is $\mathcal{O}(N^2)$
- For a given value of `ind`, adjacent `start` values have similar values of `present`.

Incremental

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    present = [False for _ in range(26)]
    for end in range(start, n):
      present[ord(s[end]) - ord('a')] = True
      distinct = sum(present)
      cur_len = end - start + 1
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- Now each substring is processed in constant time.
- Time complexity is $\mathcal{O}(N^2)$
- For a given value of `ind`, adjacent `start` values have similar values of `present`.
- Note that adding characters will never decrease `distinct`.

Incremental

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  for start in range(n):
    present = [False for _ in range(26)]
    for end in range(start, n):
      present[ord(s[end]) - ord('a')] = True
      distinct = sum(present)
      cur_len = end - start + 1
      if distinct == k and cur_len > best_len:
        best_ind = start
        best_len = cur_len
  return best_ind, best_len
```

- Now each substring is processed in constant time.
- Time complexity is $\mathcal{O}(N^2)$
- For a given value of `ind`, adjacent `start` values have similar values of `present`.
- Note that adding characters will never decrease `distinct`.
- However, removing elements from the front may reduce `distinct`.

Sliding Window

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      count[c] += 1
      end += 1
      distinct = sum(x > 0 for x in count)
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    count[ord(s[start]) - ord('a')] -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

Sliding Window

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      count[c] += 1
      end += 1
      distinct = sum(x > 0 for x in count)
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    count[ord(s[start]) - ord('a')] -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

- What is the time complexity?

Sliding Window

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      count[c] += 1
      end += 1
      distinct = sum(x > 0 for x in count)
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    count[ord(s[start]) - ord('a')] -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

- What is the time complexity? It may seem quadratic at first

Sliding Window

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      count[c] += 1
      end += 1
      distinct = sum(x > 0 for x in count)
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    count[ord(s[start]) - ord('a')] -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

- What is the time complexity? It may seem quadratic at first
- Each element gets added and removed once, so $\mathcal{O}(N)$.

Sliding Window

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      count[c] += 1
      end += 1
      distinct = sum(x > 0 for x in count)
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    count[ord(s[start]) - ord('a')] -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

- What is the time complexity? It may seem quadratic at first
- Each element gets added and removed once, so $\mathcal{O}(N)$.
- Lets introduce C, the number of different symbols possible.

Sliding Window

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      count[c] += 1
      end += 1
      distinct = sum(x > 0 for x in count)
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    count[ord(s[start]) - ord('a')] -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

- What is the time complexity? It may seem quadratic at first
- Each element gets added and removed once, so $\mathcal{O}(N)$.
- Lets introduce C, the number of different symbols possible.
- The time complexity is actually $\mathcal{O}(NC)$, but we can do better!

Sliding Window Improved

```
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      if count[c] == 0:
        distinct += 1
      count[c] += 1
      end += 1
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    c = ord(s[start]) - ord('a')
    count[c] -= 1
    if count[c] == 0:
      distinct -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

Sliding Window Improved

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      if count[c] == 0:
        distinct += 1
      count[c] += 1
      end += 1
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    c = ord(s[start]) - ord('a')
    count[c] -= 1
    if count[c] == 0:
      distinct -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

- Now adding/removing an element is $\mathcal{O}(1)$.

Sliding Window Improved

```python
def distinct_k(n, k, s):
  best_ind, best_len = -1, -1
  start, end, distinct = 0, 0, 0
  count = [0 for _ in range(26)]
  while start < n:
    while end < n:
      c = ord(s[end]) - ord('a')
      if distinct == k and count[c] == 0:
        break
      if count[c] == 0:
        distinct += 1
      count[c] += 1
      end += 1
    cur_len = end - start
    if distinct == k and cur_len > best_len:
      best_ind = start
      best_len = cur_len
    c = ord(s[start]) - ord('a')
    count[c] -= 1
    if count[c] == 0:
      distinct -= 1
    start += 1
    distinct = sum(x > 0 for x in count)
  return best_ind, best_len
```

- Now adding/removing an element is $\mathcal{O}(1)$.
- The time complexity is now $\mathcal{O}(N + C)$.

General Method

- This method is applicable when working with substrings or subarrays.

General Method

- This method is applicable when working with substrings or subarrays.
- The data has to be contiguous, or in other words, no gaps between selected elements.

General Method

- This method is applicable when working with substrings or subarrays.
- The data has to be contiguous, or in other words, no gaps between selected elements.
- Usually you want the maximal or the minimal window fulfilling a certain condition.

General Method

- Suppose that your current window is $[i, j)$ which are both initialized as 0.

General Method

- Suppose that your current window is $[i, j)$ which are both initialized as 0.
- Define an operation `add` which adds a_j to your subarray, finally increasing j by 1.

General Method

- Suppose that your current window is $[i, j)$ which are both initialized as 0.
- Define an operation `add` which adds a_j to your subarray, finally increasing j by 1.
- Define an operation `remove` which removes a_i from your subarray, finally increasing i by 1.

General Method

- Suppose that your current window is $[i, j)$ which are both initialized as 0.
- Define an operation `add` which adds a_j to your subarray, finally increasing j by 1.
- Define an operation `remove` which removes a_i from your subarray, finally increasing i by 1.
- Step 1: If performing `add` does not break your condition, perform it and repeat step 1. Otherwise go to step 2.

General Method

- Suppose that your current window is $[i, j)$ which are both initialized as 0.
- Define an operation `add` which adds a_j to your subarray, finally increasing j by 1.
- Define an operation `remove` which removes a_i from your subarray, finally increasing i by 1.
- Step 1: If performing `add` does not break your condition, perform it and repeat step 1. Otherwise go to step 2.
- Step 2: Check if the current window is a better answer and possibly update. Then go to step 3.

General Method

- Suppose that your current window is $[i, j)$ which are both initialized as 0.
- Define an operation `add` which adds a_j to your subarray, finally increasing j by 1.
- Define an operation `remove` which removes a_i from your subarray, finally increasing i by 1.
- Step 1: If performing `add` does not break your condition, perform it and repeat step 1. Otherwise go to step 2.
- Step 2: Check if the current window is a better answer and possibly update. Then go to step 3.
- Step 3: Perform `remove` and go to step 1.

General Method

- Suppose that your current window is $[i, j)$ which are both initialized as 0.
- Define an operation `add` which adds a_j to your subarray, finally increasing j by 1.
- Define an operation `remove` which removes a_i from your subarray, finally increasing i by 1.
- Step 1: If performing `add` does not break your condition, perform it and repeat step 1. Otherwise go to step 2.
- Step 2: Check if the current window is a better answer and possibly update. Then go to step 3.
- Step 3: Perform `remove` and go to step 1.
- Time complexity is $\mathcal{O}(N \cdot (X + Y))$ where X and Y are the cost of `add` and `remove`, respectively.

Heap

Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.

Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.
- Heaps are implemented in most standard libraries in the forms of priority queues.

Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.
- Heaps are implemented in most standard libraries in the forms of priority queues.
- A heap is nothing but a binary tree satisfying *the heap condition*.

Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.
- Heaps are implemented in most standard libraries in the forms of priority queues.
- A heap is nothing but a binary tree satisfying *the heap condition*.
- The heap condition (for a min heap) says that the value of any given node is not greater than that of its chilren.

Heaps

- Since arrays are linear, we want to smush this binary tree into an array for the implementation.

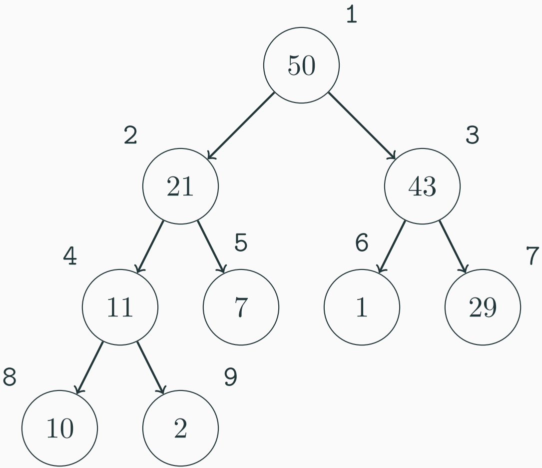
Heaps

- Since arrays are linear, we want to smush this binary tree into an array for the implementation.
- We can do this by putting the root at index 1. Then the children of item at index i are simply at $2i$ and $2i + 1$. The parent of any item $i > 1$ is then $\left\lfloor \frac{i}{2} \right\rfloor$.

Heaps

- Since arrays are linear, we want to smush this binary tree into an array for the implementation.
- We can do this by putting the root at index 1. Then the children of item at index i are simply at $2i$ and $2i+1$. The parent of any item $i > 1$ is then $\left\lfloor \frac{i}{2} \right\rfloor$.
- We could do this using raw arrays (then index 0 can be used to store its size), but the examples will be given in C++ using vectors.

Heaps



ARRAY: [SIZE, 50, 21, 43, 11, 7, 1, 29, 10, 2]

Heaps

- Items can be inserted by pushing them to the back and fixing the heap condition upwards from them.

Heaps

- Items can be inserted by pushing them to the back and fixing the heap condition upwards from them.
- Items can be deleted by replacing the smallest value with a leaf and then fixing the heap condition downwards.

Heaps

- Items can be inserted by pushing them to the back and fixing the heap condition upwards from them.
- Items can be deleted by replacing the smallest value with a leaf and then fixing the heap condition downwards.
- Let us see how this would look in C++.

C++ implementation (min-heap)

```cpp
template<typename T> struct Heap {
    vector<T> h; Heap() : h(1) { }
    constexpr size_t size() { return h.size() - 1; }
    constexpr T peek() { return h[1]; }
    void swim(size_t i) {
        while(i != 1 && h[i] < h[i / 2]) {
            swap(h[i], h[i / 2]);
            i /= 2; } }
    void sink(size_t i) {
        while(true) {
            size_t mn = i;
            if(2 * i + 1 < h.size() && h[mn] > h[2 * i + 1]) mn = 2 * i + 1;
            if(2 * i  < h.size() && h[mn] > h[2 * i]) mn = 2 * i;
            if(mn != i) swap(h[i], h[mn]), i = mn;
            else break; } }
    void pop() {
        h[1] = h.back();
        h.pop_back(); sink(1); }
    void push(T x) {
        h.push_back(x);
        swim(h.size() - 1); } };
```

Heaps

- We note that peek and size run in $\mathcal{O}(1)$ while all other operations run in $\mathcal{O}(log(n))$.

Heaps

- We note that peek and size run in $\mathcal{O}(1)$ while all other operations run in $\mathcal{O}(log(n))$.
- This implementation isn't any better than the standard library one in C++.

Heaps

- We note that peek and size run in $\mathcal{O}(1)$ while all other operations run in $\mathcal{O}(log(n))$.
- This implementation isn't any better than the standard library one in C++.
- We provide it for demonstration of representing binary trees with an array.

Union-Find

Union-Find

- We have n items

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.
- Operation `find(x)` finds the representative of the set x is in

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.
- Operation `find(x)` finds the representative of the set x is in
- Operation `union(x, y)` unions the sets of which x and y are members.

Union-Find

- It is generally initialized with all items being in their own set.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `union(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `union(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `union(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `union(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `union(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `union(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `union(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `union(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `union(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.
- `union(1, 4)` finally results in $\{\{1, 2, 3, 4, 5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `union(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `union(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `union(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.
- `union(1, 4)` finally results in $\{\{1, 2, 3, 4, 5\}\}$.
- At any given point `find(x)` returns some value in the same set as x.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `union(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `union(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `union(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.
- `union(1, 4)` finally results in $\{\{1, 2, 3, 4, 5\}\}$.
- At any given point `find(x)` returns some value in the same set as x.
- The important bit is that `find(x)` returns the same value for all elements of the same set, the representative.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.
- If a value has no parent, we can denote this somehow, make it its own parent, give it the value -1, exactly what we do is not important.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.
- If a value has no parent, we can denote this somehow, make it its own parent, give it the value -1, exactly what we do is not important.
- To get the representative of x we go to the parent of our current item (starting at x) until the item has no parent.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.
- If a value has no parent, we can denote this somehow, make it its own parent, give it the value -1, exactly what we do is not important.
- To get the representative of x we go to the parent of our current item (starting at x) until the item has no parent.
- Then to unite x, y we simply make the representative of x the parent of the representative of y.

Naïve Union-Find implementation

```
struct union_find {
    vector<int> parent;
    union_find(int n) {
        parent = vector<int>(n);
        for(int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {
        return parent[x] == x ? x : find(parent[x]);
    }
    void unite(int x, int y) {
        parent[find(x)] = find(y);
    }
};
```

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.
- One method is to use what is known as small-to-large merging, where the smaller group's leader is made to point to the larger group's leader.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.
- One method is to use what is known as small-to-large merging, where the smaller group's leader is made to point to the larger group's leader.
- This ensures the height increases by 1 as a group's size doubles, resulting in $\mathcal{O}(\log n)$ complexity.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.
- One method is to use what is known as small-to-large merging, where the smaller group's leader is made to point to the larger group's leader.
- This ensures the height increases by 1 as a group's size doubles, resulting in $\mathcal{O}(\log n)$ complexity.
- We can also do this by flattening the chain each time we query `find`, so the amortized complexity becomes good.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.
- One method is to use what is known as small-to-large merging, where the smaller group's leader is made to point to the larger group's leader.
- This ensures the height increases by 1 as a group's size doubles, resulting in $\mathcal{O}(\log n)$ complexity.
- We can also do this by flattening the chain each time we query `find`, so the amortized complexity becomes good.
- Here the worst case is still $\mathcal{O}(n)$ but the amortized complexity is $\mathcal{O}(\alpha(n))$ which may as well be a constant, as it is < 5 for n equal to the number of atoms in the observable universe.

Path compressed Union-Find implementation

```
struct union_find {
    vector<int> parent;
    union_find(int n) {
        parent = vector<int>(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {
        if(parent[x] == x) return x;
        return parent[x] = find(parent[x]);
    }
    void unite(int x, int y) {
        parent[find(x)] = find(y);
    }
};
```

Union-Find applications

- Union-Find maintains a collection of disjoint sets

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
    - Number of different sets currently

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
  - Number of different sets currently
  - Current size of the set containing x

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
  - Number of different sets currently
  - Current size of the set containing x
  - An iterable list of all elements of the set containing x

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
  - Number of different sets currently
  - Current size of the set containing x
  - An iterable list of all elements of the set containing x
- When tracking size you can use it to always perform small-to-large merges for $\mathcal{O}(\log n)$ time complexity.

Example problem: Skolavslutningen

- https://open.kattis.com/problems/skolavslutningen

Range Queries

Range queries

- We have an array A of size n.

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \dots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \dots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\mathrm{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\text{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$
- We want to answer these queries efficiently, or in other words, without looking through all elements.

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\mathrm{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$
- We want to answer these queries efficiently, or in other words, without looking through all elements.
- Sometimes we also want to update elements.

Range sum on constant array

- Let's look at range sums on a constant array

Range sum on constant array

- Let's look at range sums on a constant array
- How do we support these queries efficiently?

Range sum on constant array

- Let's look at range sums on a constant array
- How do we support these queries efficiently?
- Simplification: only support queries of the form $\mathrm{sum}(0, j)$

Range sum on constant array

- Let's look at range sums on a constant array
- How do we support these queries efficiently?
- Simplification: only support queries of the form $\mathrm{sum}(0, j)$
- Notice that $\mathrm{sum}(i, j) = \mathrm{sum}(0, j) - \mathrm{sum}(0, i - 1)$

Range sum on a static array

- So we're only interested in prefix sums

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1						

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1					

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8				

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16			

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21		

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

- $O(n)$ time to preprocess

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

- $O(n)$ time to preprocess
- $O(1)$ time each query

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

- $O(n)$ time to preprocess
- $O(1)$ time each query
- Can we support updating efficiently?

Range sum on a static array

- So we're only interested in prefix sums
- But there are only n of them...
- Just compute them all once in the beginning

1	0	7	8	5	9	3
1	1	8	16	21	30	33

- $O(n)$ time to preprocess
- $O(1)$ time each query

- Can we support updating efficiently? No, at least not without modification

Generalizing

- This works on any invertible function.

Generalizing

- This works on any invertible function.
- If we want the product we can store the products and use $\mathrm{mul}(i, j) = \mathrm{mul}(0, j)/\mathrm{mul}(0, i-1)$.

Generalizing

- This works on any invertible function.
- If we want the product we can store the products and use $\mathrm{mul}(i, j) = \mathrm{mul}(0, j)/\mathrm{mul}(0, i-1)$.
- This also works for multidimensional arrays, but the math is more involved.

Generalizing

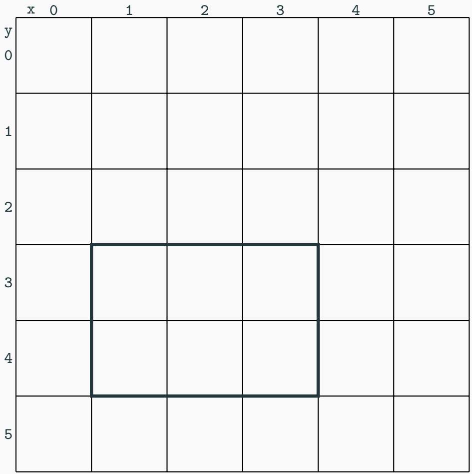
- This works on any invertible function.
- If we want the product we can store the products and use $\mathrm{mul}(i, j) = \mathrm{mul}(0, j)/\mathrm{mul}(0, i-1)$.
- This also works for multidimensional arrays, but the math is more involved.
- We let $\mathrm{sum}(x_i, x_j, y_i, y_j)$ denote the query for the sum from x_i to x_j along the x-dimension, and the same for y.

Generalizing

- This works on any invertible function.
- If we want the product we can store the products and use $\mathrm{mul}(i, j) = \mathrm{mul}(0, j)/\mathrm{mul}(0, i-1)$.
- This also works for multidimensional arrays, but the math is more involved.
- We let $\mathrm{sum}(x_i, x_j, y_i, y_j)$ denote the query for the sum from x_i to x_j along the x-dimension, and the same for y.
- Then the formula becomes

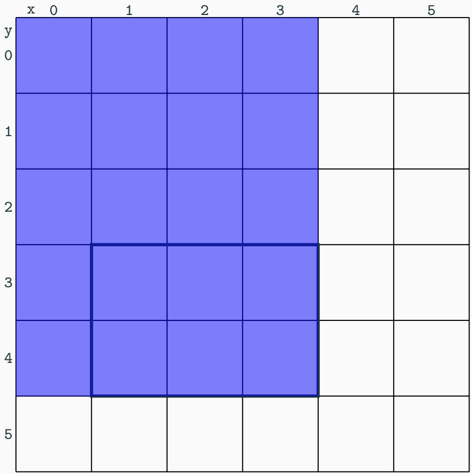
$$\begin{aligned}\mathrm{sum}(x_i, x_j, y_i, y_j) &= \mathrm{sum}(0, x_j, 0, y_j)\\ &- \mathrm{sum}(0, x_{i-1}, 0, y_j)\\ &- \mathrm{sum}(0, x_j, 0, y_{i-1})\\ &+ \mathrm{sum}(0, x_{i-1}, 0, y_{i-1})\end{aligned}$$

2D sum



`query(1, 3, 3, 4)`

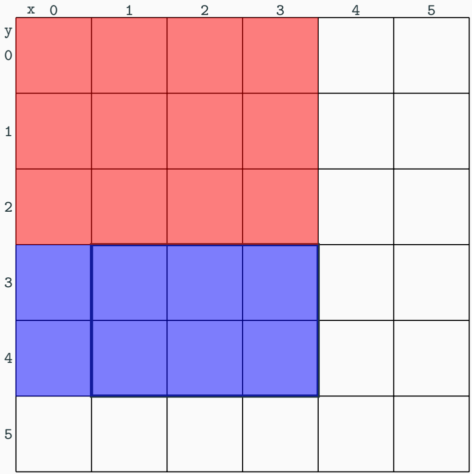
2D sum



`query(1, 3, 3, 4)`

`query(0, 3, 0, 4)`

2D sum

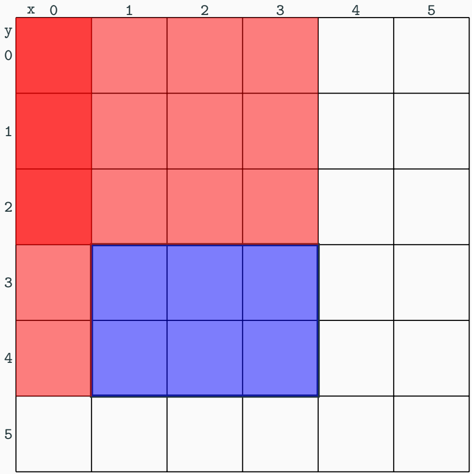


query(1, 3, 3, 4)

query(0, 3, 0, 4)

query(0, 4, 0, 2)

2D sum



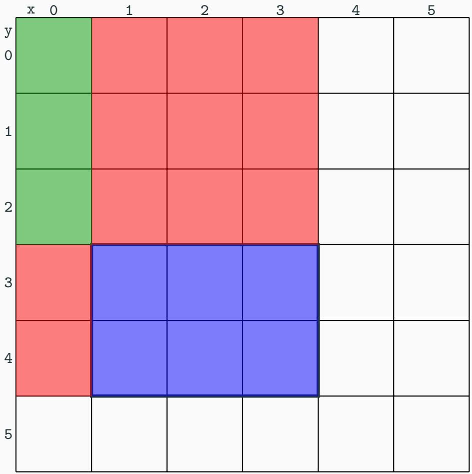
query(1, 3, 3, 4)

query(0, 3, 0, 4)

query(0, 4, 0, 2)

query(0, 0, 0, 4)

2D sum




```
query(1, 3, 3, 4)
query(0, 3, 0, 4)
query(0, 4, 0, 2)
query(0, 0, 0, 4)
query(0, 0, 0, 2)
```

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
    - change the array element

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
    - change the array element
    - recompute corresponding bucket

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
    - change the array element
    - recompute corresponding bucket
- Time complexity: $O(k)$

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to “jump” over intervals of size k

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets and 2 buckets of size k

First attempt: Buckets

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets and 2 buckets of size k
- Time complexity: $O(n/k + k)$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Querying in $O(n/k + k)$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
  - Updating in $O(\sqrt{n})$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
  - Updating in $O(\sqrt{n})$
  - Querying in $O(n/\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$

Buckets: Choosing k

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
  - Updating in $O(\sqrt{n})$
  - Querying in $O(n/\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$
- Also known as square root decomposition, and is a very powerful technique

Example problem: Supercomputer

- https://open.kattis.com/problems/supercomputer

A better approach?

- Now we know how to do these queries in $O(\sqrt{n})$

A better approach?

- Now we know how to do these queries in $O(\sqrt{n})$
- May be too slow if n is large and many queries

A better approach?

- Now we know how to do these queries in $O(\sqrt{n})$
- May be too slow if n is large and many queries

- Can we do better?

Second attempt: Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.

Second attempt: Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.
- Then each internal vertex is the sum of the values below it.

Second attempt: Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.
- Then each internal vertex is the sum of the values below it.
- Then we have $\mathcal{O}(n)$ nodes and each query can be pieced together from $\mathcal{O}(\log(n))$ node values.

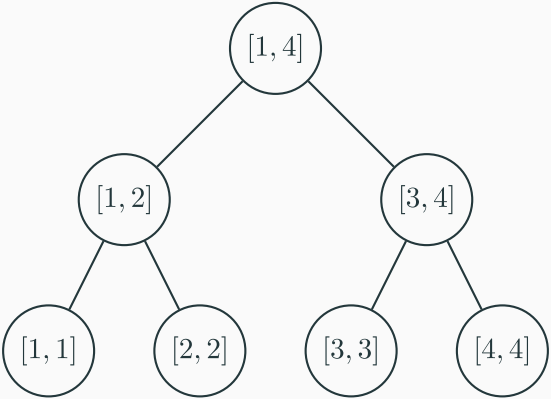
Second attempt: Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.
- Then each internal vertex is the sum of the values below it.
- Then we have $\mathcal{O}(n)$ nodes and each query can be pieced together from $\mathcal{O}(\log(n))$ node values.
- We travel down the tree looking for the left and right end points, adding intervals that are completely inside our query range.

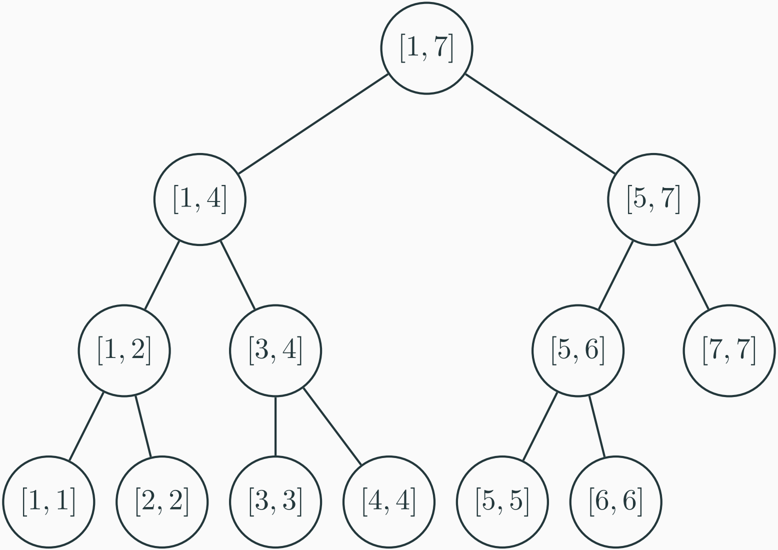
Second attempt: Segment Tree

- We create a perfect binary tree where the leaves are the elements of the array.
- Then each internal vertex is the sum of the values below it.
- Then we have $\mathcal{O}(n)$ nodes and each query can be pieced together from $\mathcal{O}(\log(n))$ node values.
- We travel down the tree looking for the left and right end points, adding intervals that are completely inside our query range.
- When we update a value we only need to update the parents of that node up to the root, at most $\mathcal{O}(\log(n))$ nodes.

Drawn Segment Tree, $n = 4$



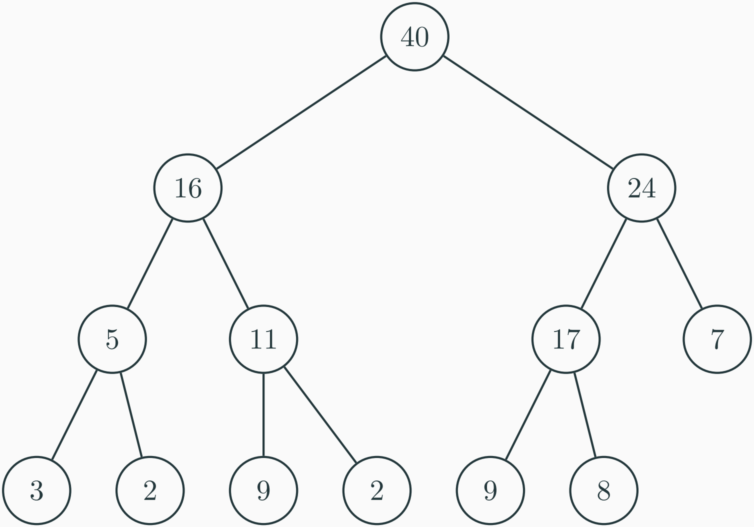
Drawn Segment Tree, $n = 7$



Segment Tree - Code

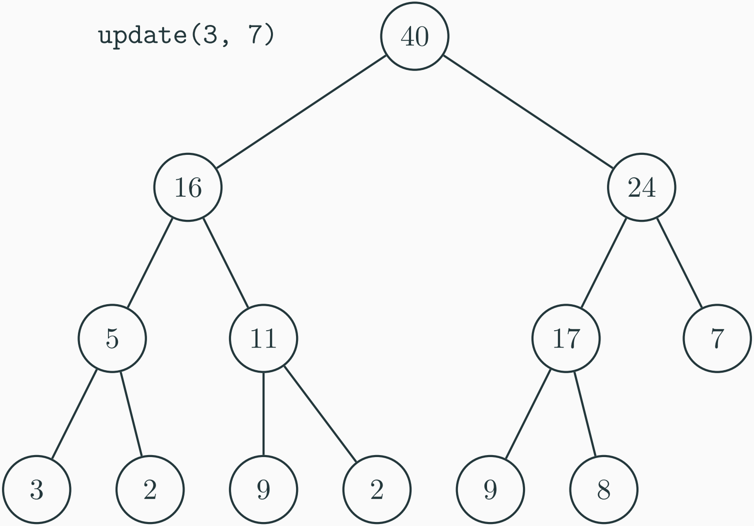
```
struct segment_tree {
    segment_tree *left, *right;
    int from, to, value;
    segment_tree(int from, int to)
        : from(from), to(to), left(NULL), right(NULL), value(0) { }
};

segment_tree* build(const vector<int> &arr, int l, int r) {
    if (l > r) return NULL;
    segment_tree *res = new segment_tree(l, r);
    if (l == r) {
        res->value = arr[l];
    } else {
        int m = (l + r) / 2;
        res->left = build(arr, l, m);
        res->right = build(arr, m + 1, r);
        if (res->left != NULL) res->value += res->left->value;
        if (res->right != NULL) res->value += res->right->value;
    }
    return res;
}
```

Updates

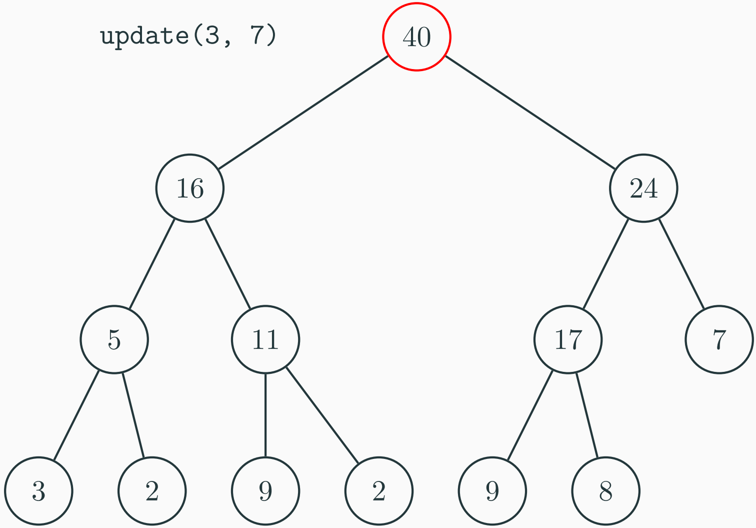


Updates

`update(3, 7)`

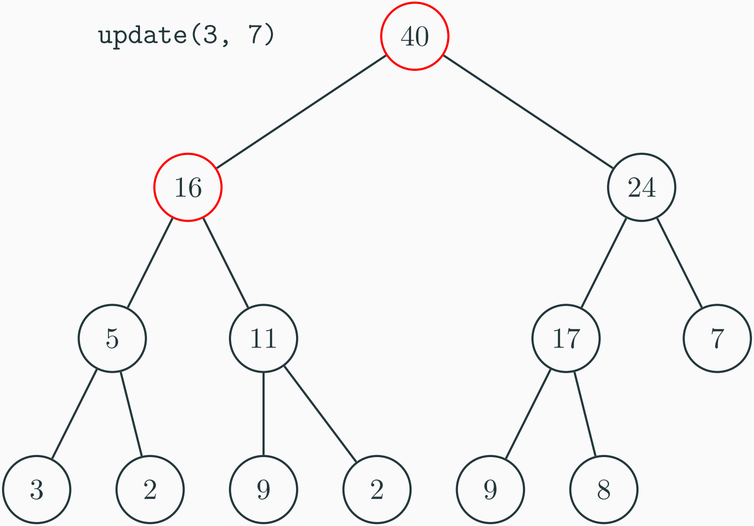


Updates



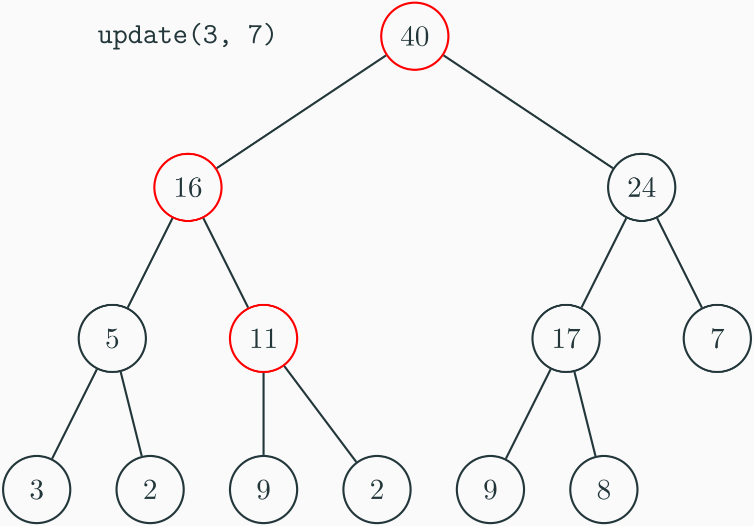
Updates

`update(3, 7)`

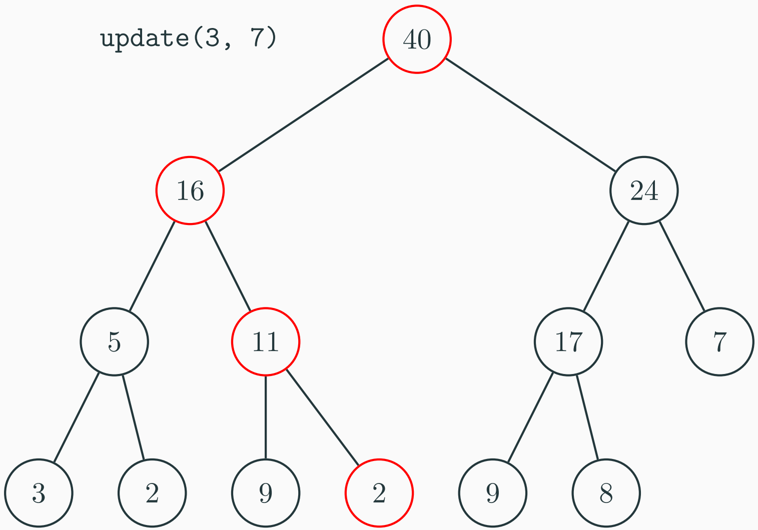


Updates

`update(3, 7)`

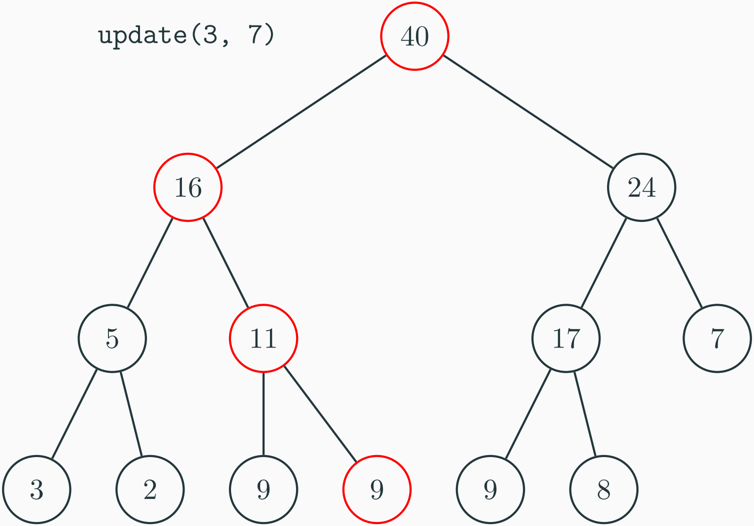


Updates



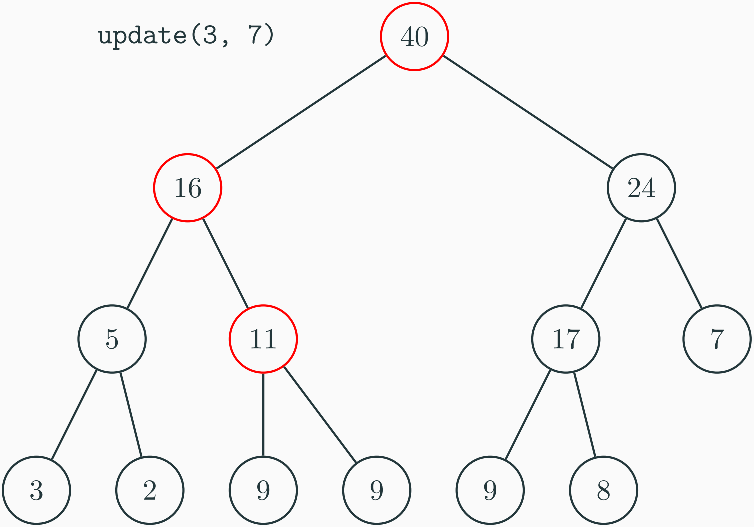
Updates

`update(3, 7)`



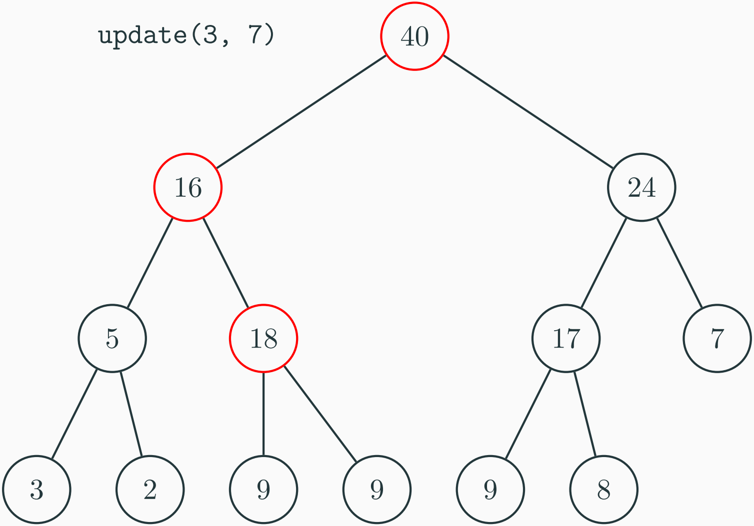
Updates

`update(3, 7)`

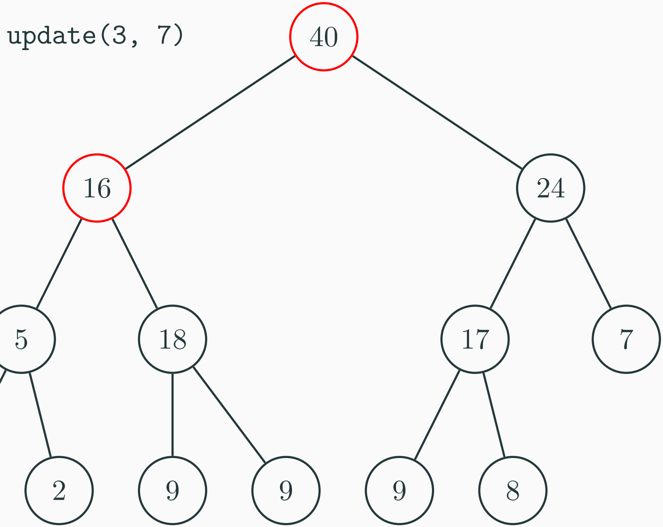


Updates

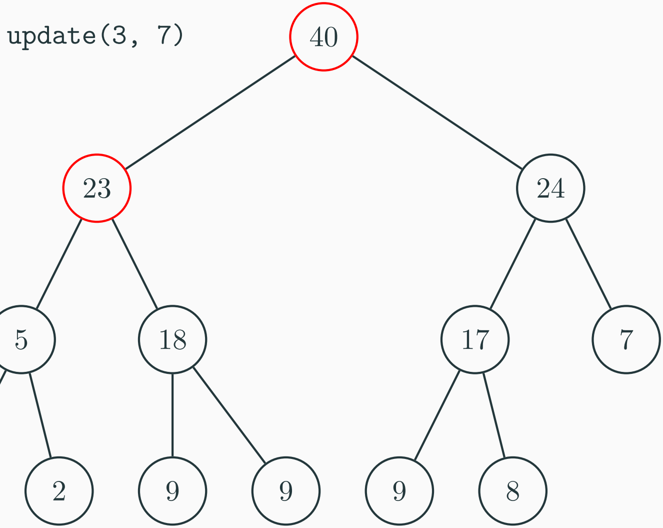
`update(3, 7)`



Updates

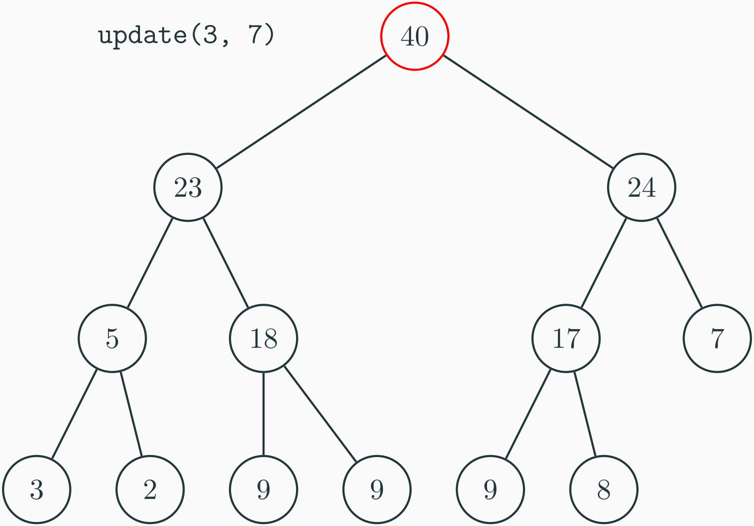


Updates



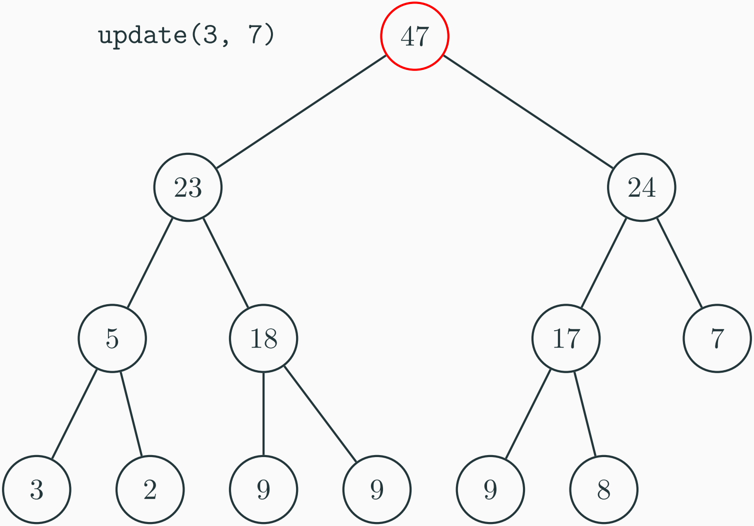
Updates

`update(3, 7)`

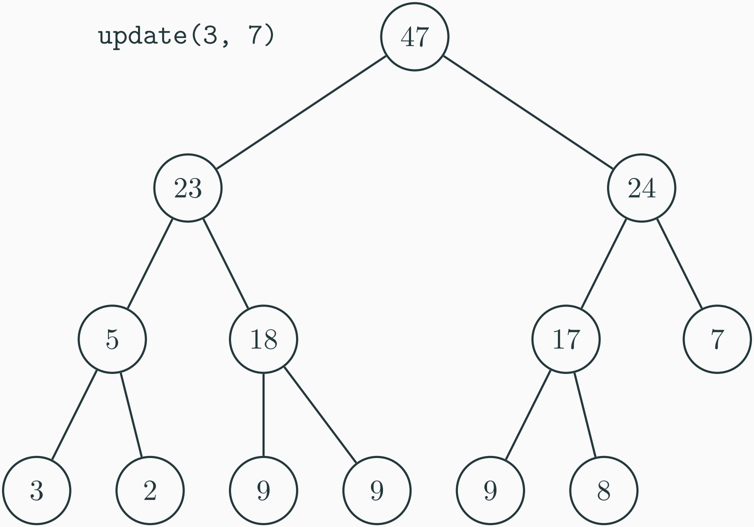


Updates

`update(3, 7)`

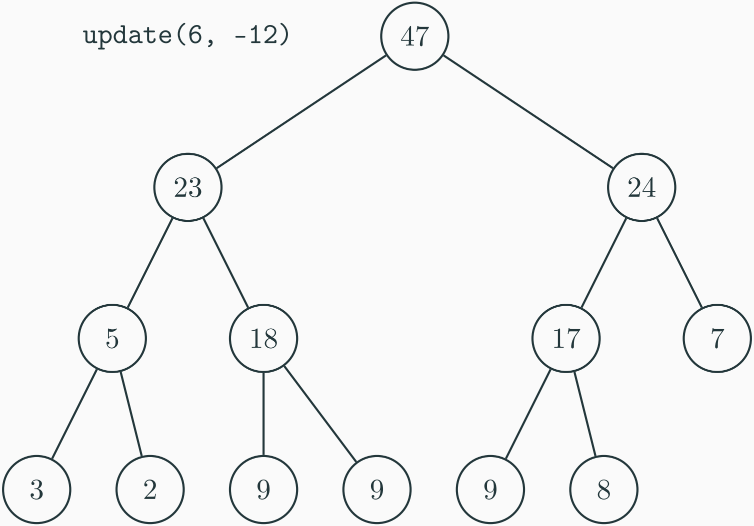


Updates

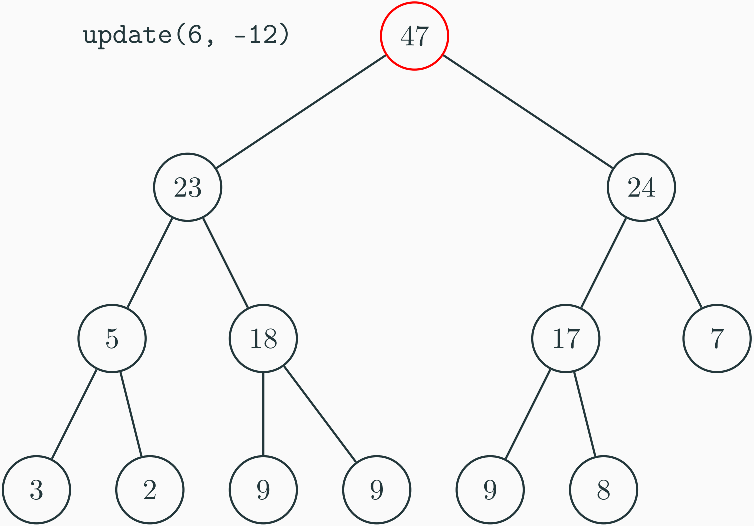


Updates

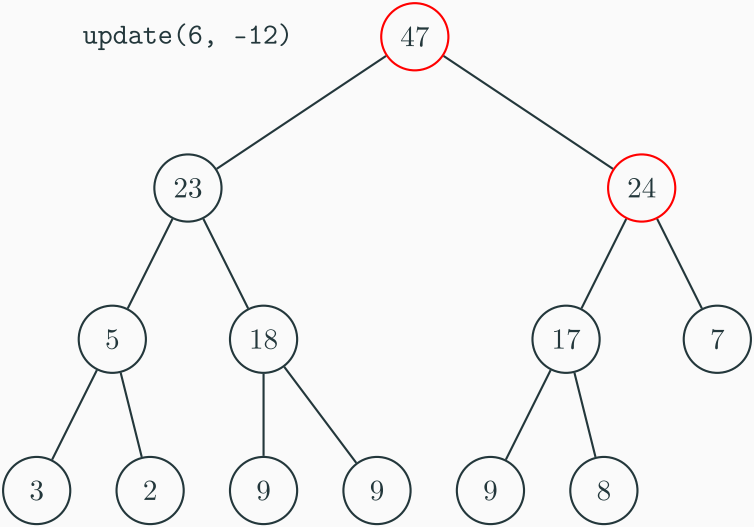
```
update(6, -12)
```



Updates

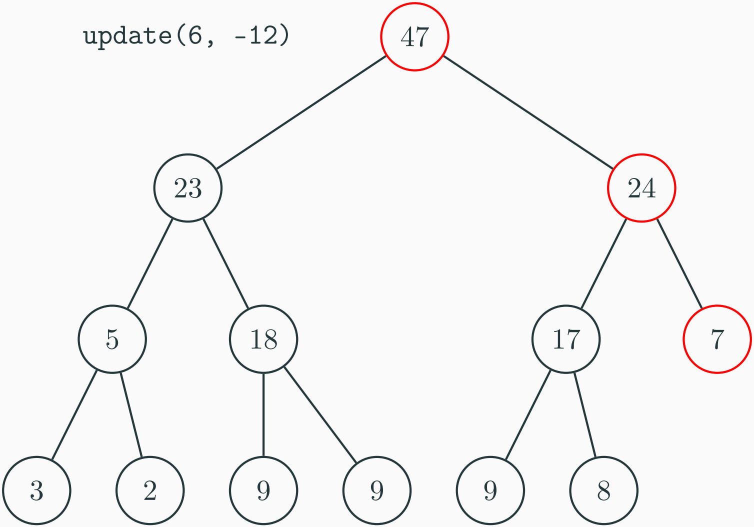


Updates

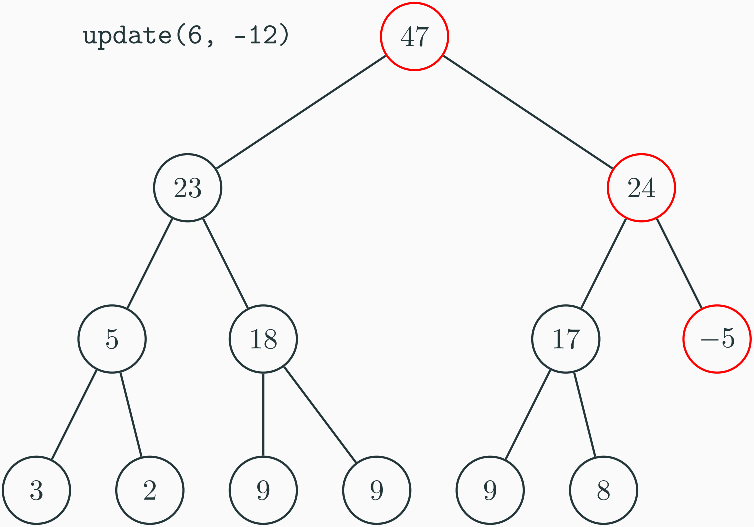


Updates

`update(6, -12)`

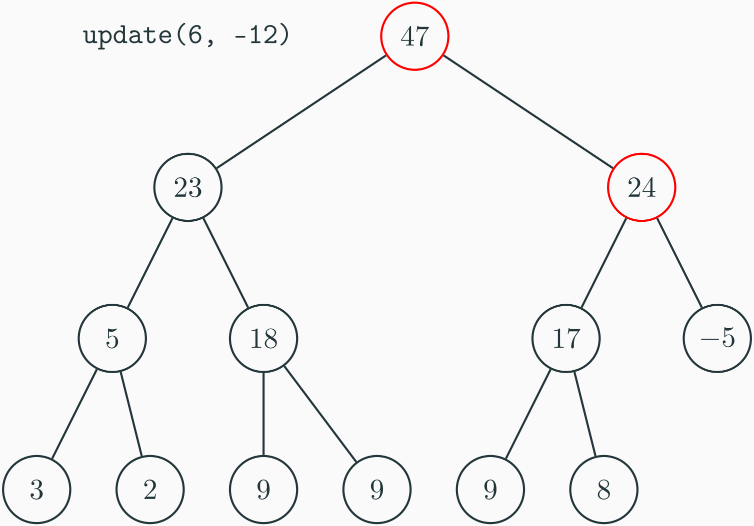


Updates



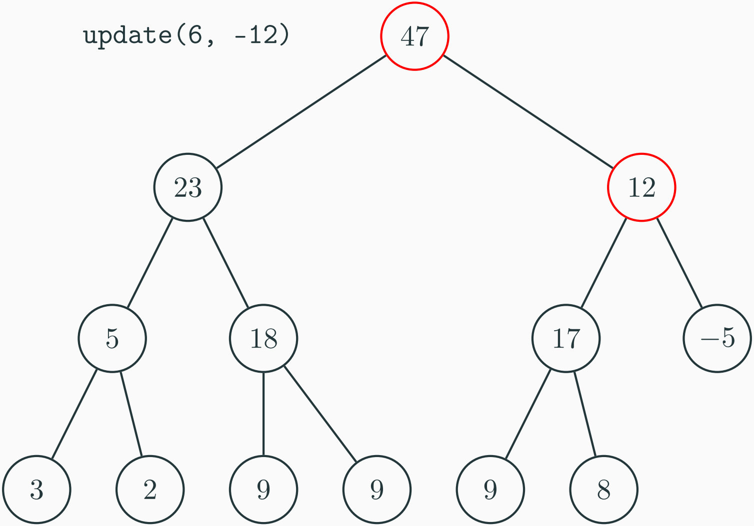
Updates

`update(6, -12)`

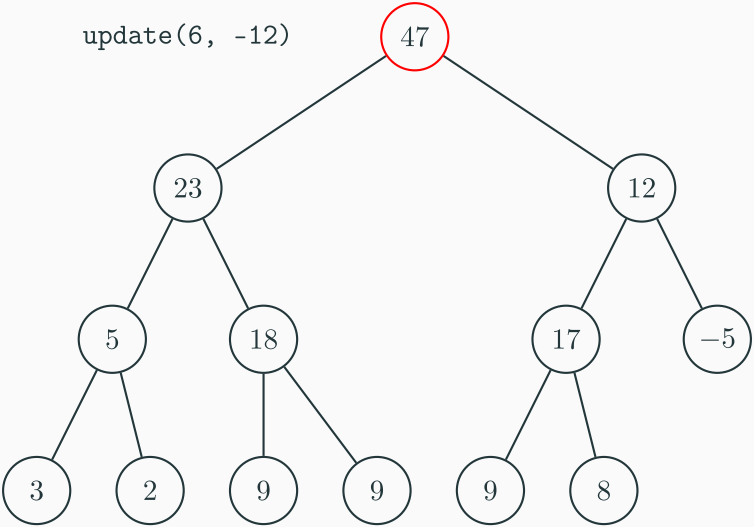


Updates

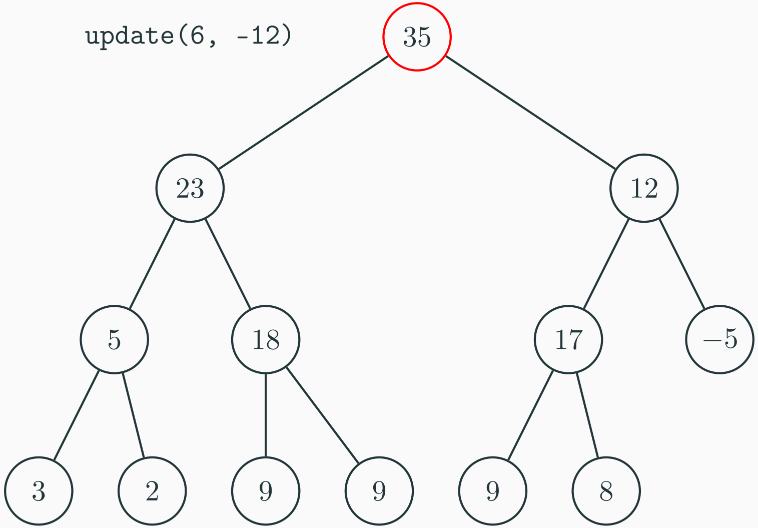
`update(6, -12)`



Updates

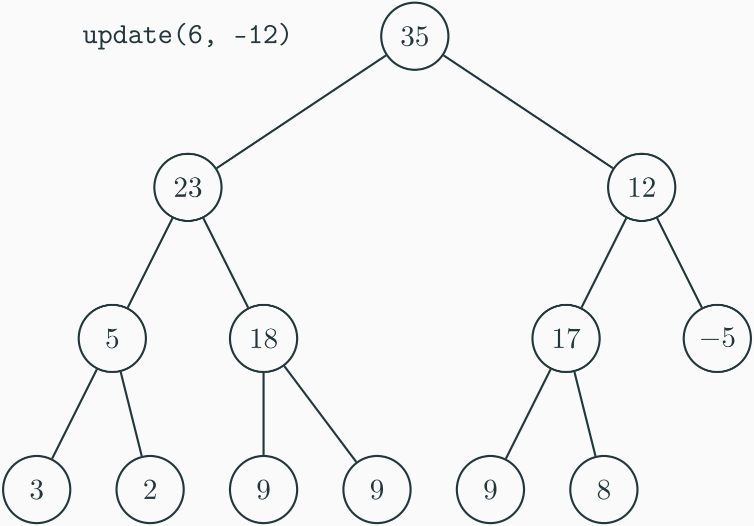


Updates

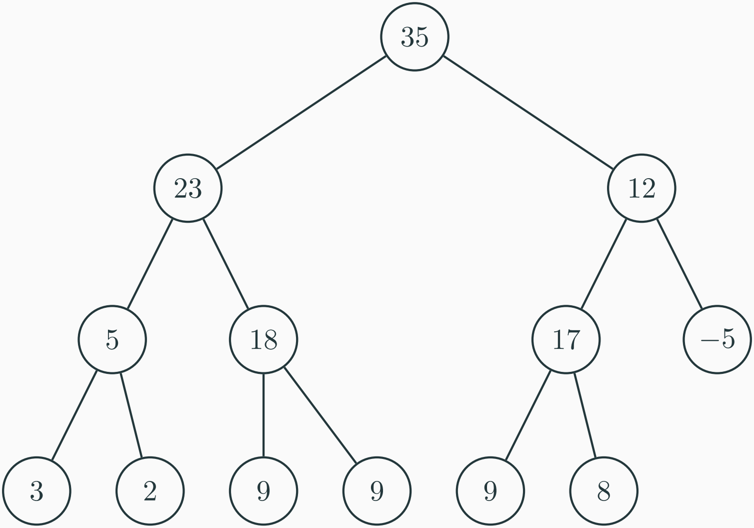


Updates

`update(6, -12)`



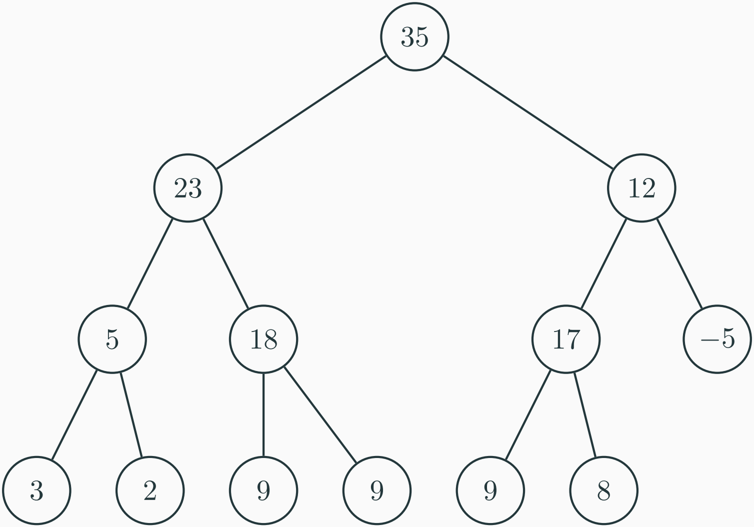
Updates



Updating a Segment Tree - Code

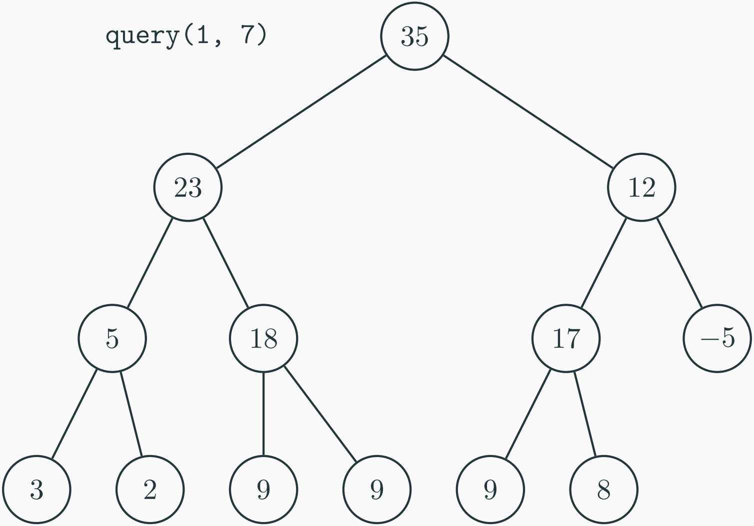
```c
int update(segment_tree *tree, int i, int val) {
    if (tree == NULL) return 0;
    if (tree->to < i) return tree->value;
    if (i < tree->from) return tree->value;
    if (tree->from == tree->to && tree->from == i) {
        tree->value = val;
    } else {
        tree->value = update(tree->left, i, val) + update(tree->right, i, val);
    }
    return tree->value;
}
```

Querying

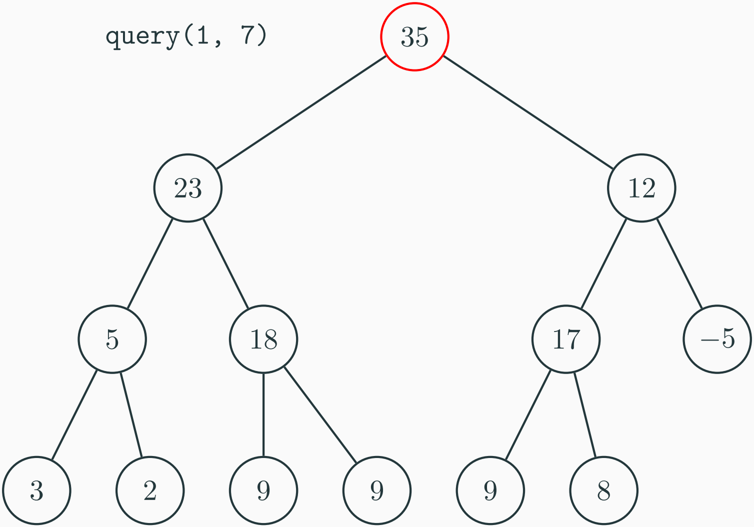


Querying

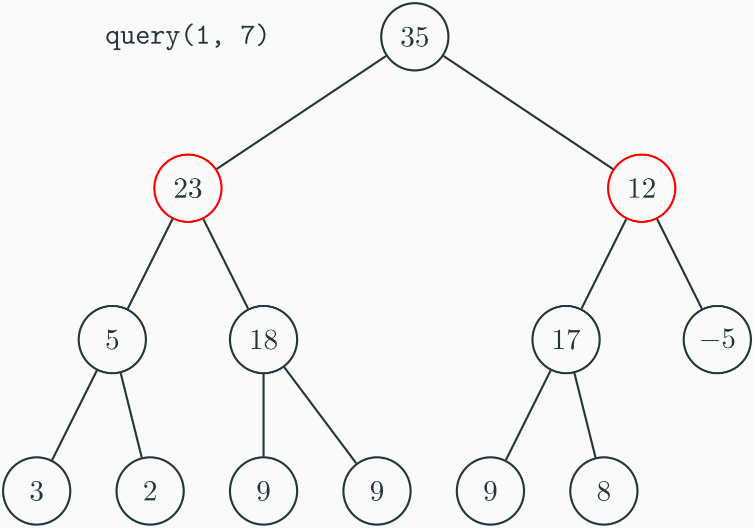
```
query(1, 7)
```



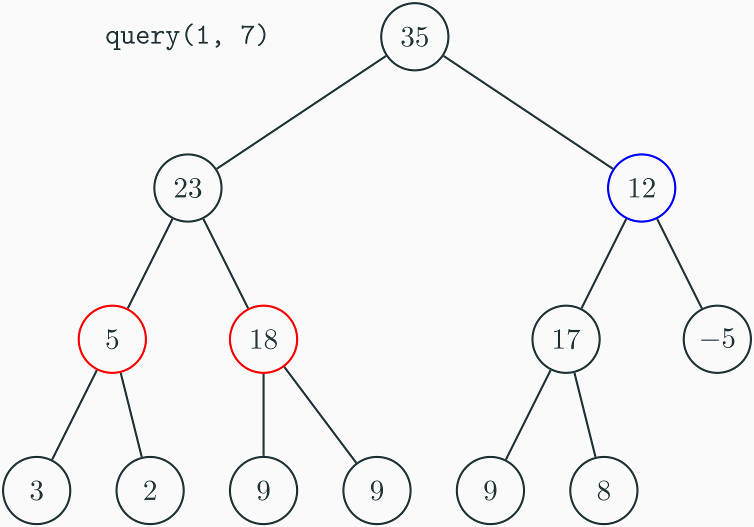
Querying



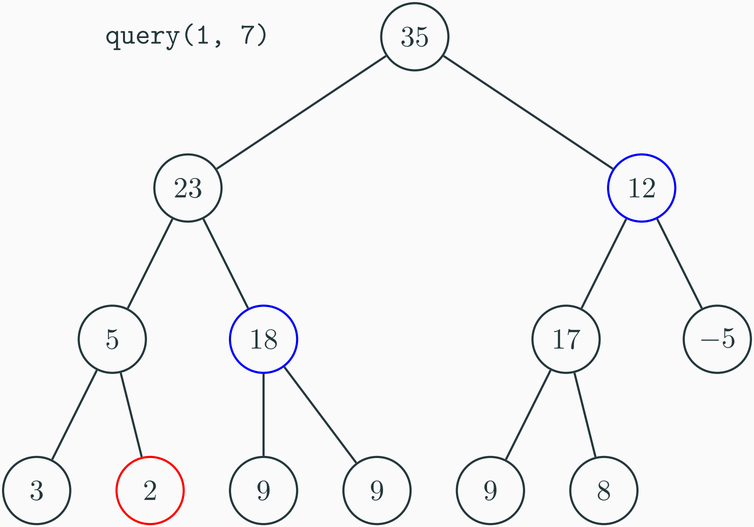
Querying



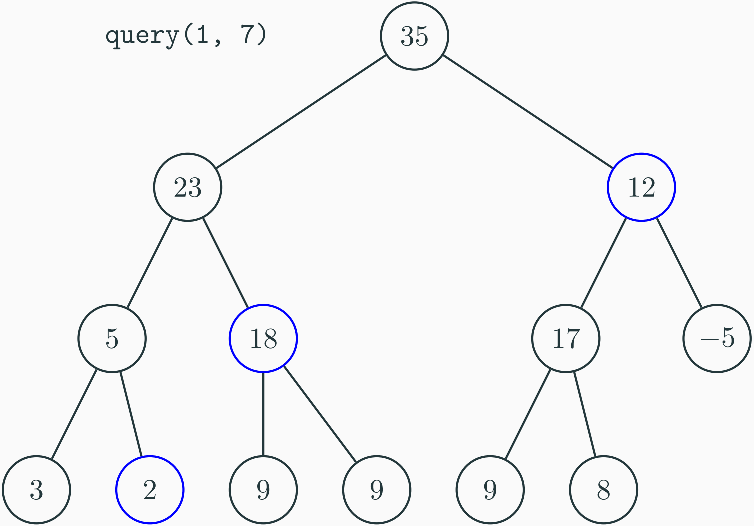
Querying



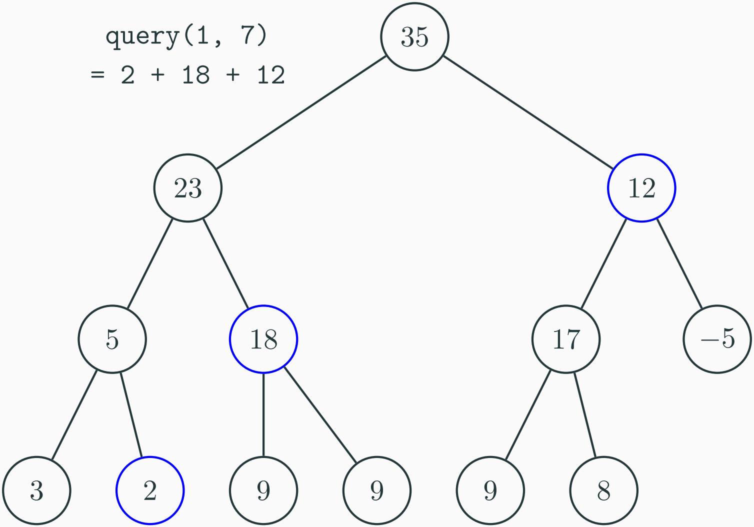
Querying



Querying

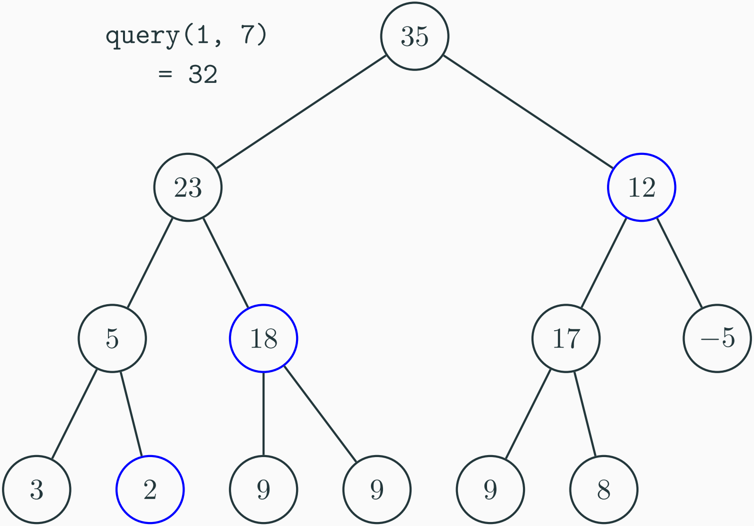


Querying



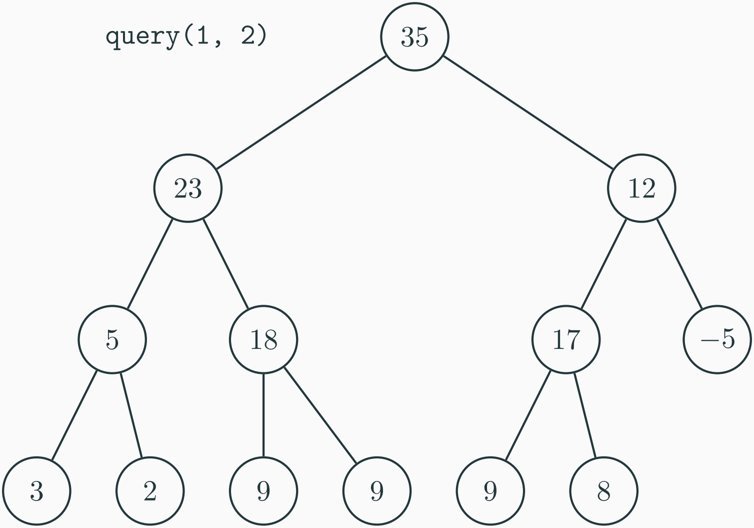
Querying

```
query(1, 7)
   = 32
```

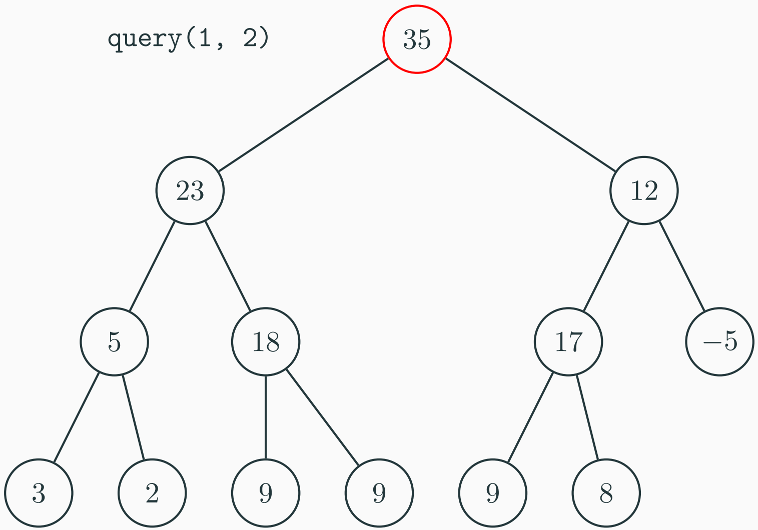


Querying

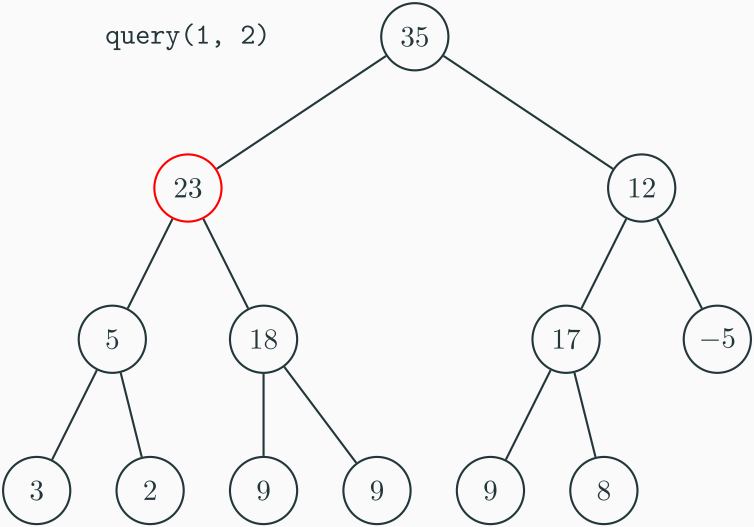
`query(1, 2)`



Querying

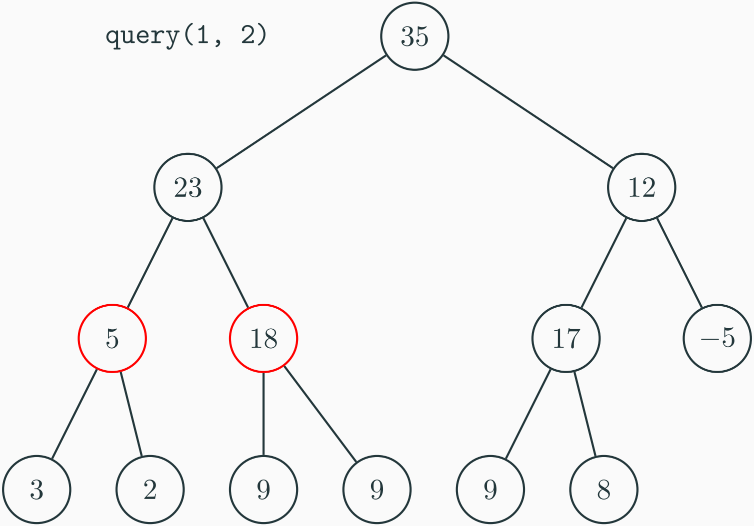


Querying

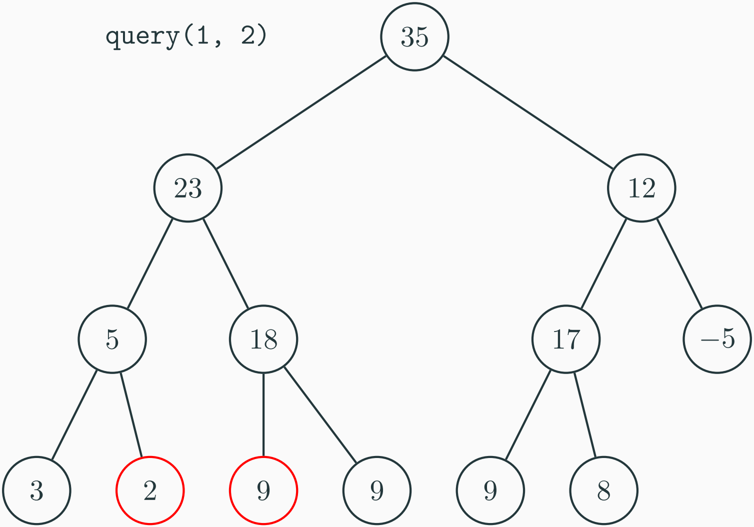


Querying

`query(1, 2)`

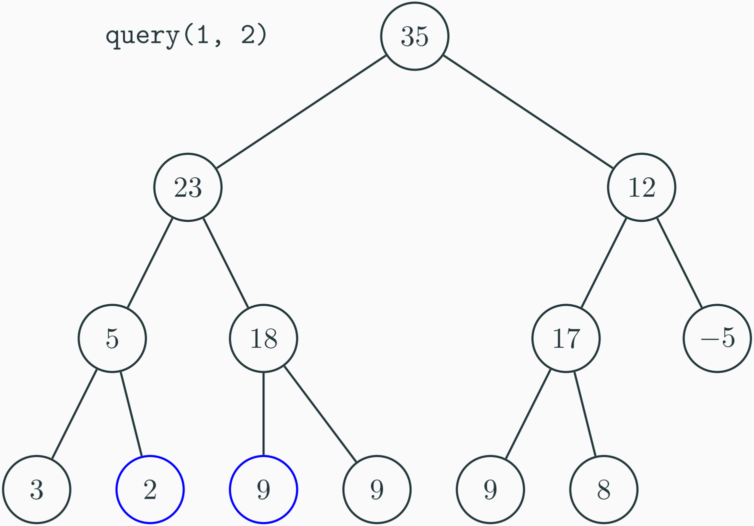


Querying

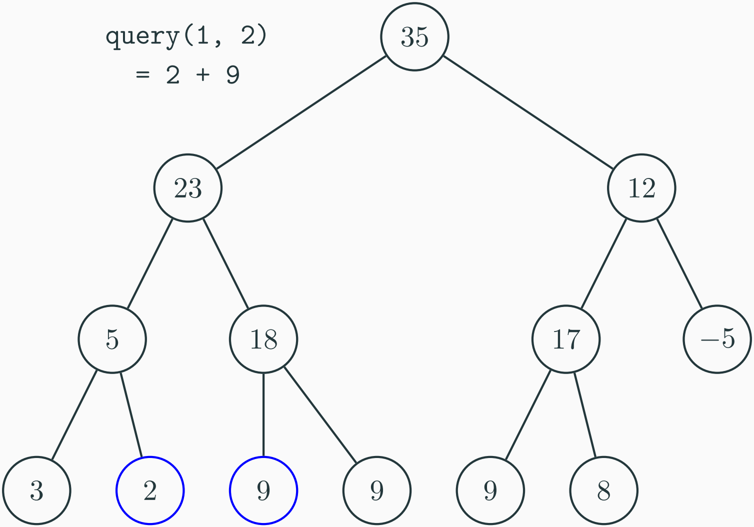


Querying

`query(1, 2)`

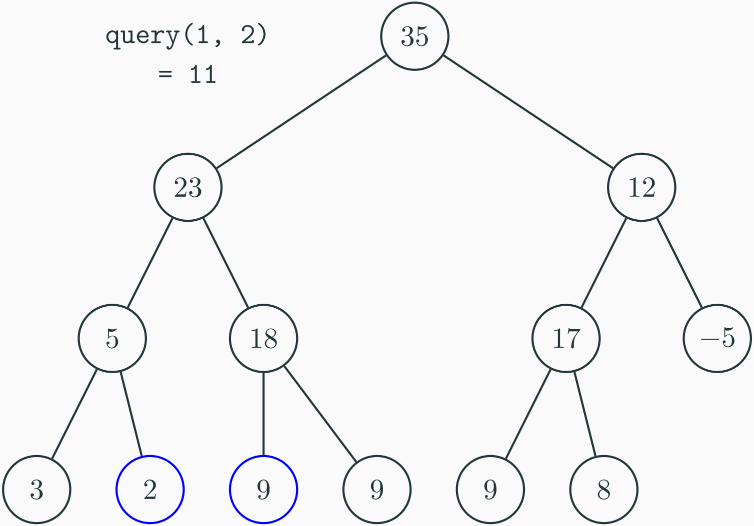


Querying

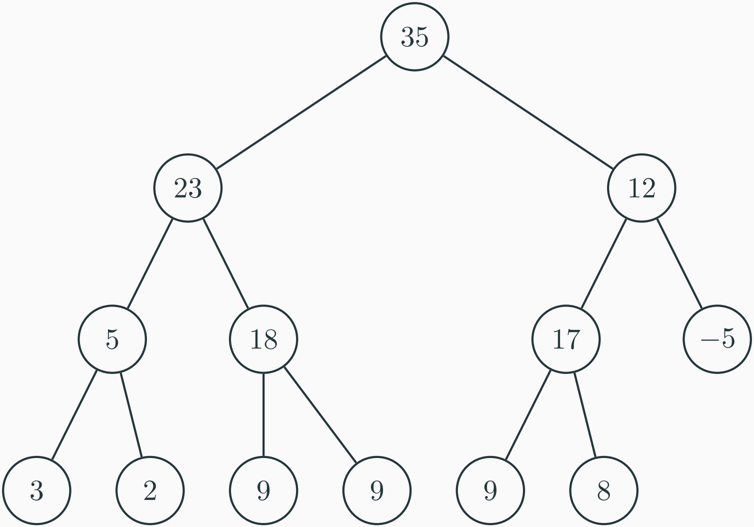


Querying

```
query(1, 2)
    = 11
```



Querying



Querying a Segment Tree - Code

```
int query(segment_tree *tree, int l, int r) {
    if (tree == NULL) return 0;
    if (l <= tree->from && tree->to <= r) return tree->value;
    if (tree->to < l) return 0;
    if (r < tree->from) return 0;
    return query(tree->left, l, r) + query(tree->right, l, r);
}
```

Segment Tree

- Simple to use Segment Trees for $\min$, $\max$, $\gcd$, and other similar operators, basically the same code.

Segment Tree

- Simple to use Segment Trees for $\min$, $\max$, $\gcd$, and other similar operators, basically the same code.
- Any associative operator will work.

Segment Tree

- Simple to use Segment Trees for $\min$, $\max$, $\gcd$, and other similar operators, basically the same code.
- Any associative operator will work.
- So any operator f such that $f(a, f(b, c)) = f(f(a, b), c)$ for all a, b, c.

Example problem: Movie Collection

- https://open.kattis.com/problems/moviecollection

Range updates

- So far, we have only allowed updates to affect a single element.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.
- Can we make use of our segmented structure to update all indices in range $[l, r]$?

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.
- Can we make use of our segmented structure to update all indices in range $[l, r]$?
- Lazy people tend to find efficient ways of doing all that **needs** to be done, but no more.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.
- Can we make use of our segmented structure to update all indices in range $[l, r]$?
- Lazy people tend to find efficient ways of doing all that **needs** to be done, but no more.
- After updating a value, there is no guarantee you will use the updated value afterwards.

Range updates

- So far, we have only allowed updates to affect a single element.
- Might want to update multiple elements simultaneously, iterating for each is expensive.
- Example: For all indices from 4 to 7 add 13.
- Can we make use of our segmented structure to update all indices in range $[l, r]$?
- Lazy people tend to find efficient ways of doing all that **needs** to be done, but no more.
- After updating a value, there is no guarantee you will use the updated value afterwards.
- Idea: Be lazy and procrastinate changes until they are needed!

Lazy propagation

- Add another variable for each node, storing the lazy value

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.
- When looking at a node, apply the lazy value to the node

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.
- When looking at a node, apply the lazy value to the node
- After applying, push the lazy value to the two child nodes

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.
- When looking at a node, apply the lazy value to the node
- After applying, push the lazy value to the two child nodes
- Reset the lazy value.

Lazy propagation

- Add another variable for each node, storing the lazy value
- Follow same steps as querying in order to update, applying the update if the node is completely within the range.
- When making updates, do not change the original data variable, but rather the lazy variable.
- When looking at a node, apply the lazy value to the node
- After applying, push the lazy value to the two child nodes
- Reset the lazy value.
- Traverse to child nodes if needed.

Code example

See implementation example, for example here.

Sparse Table

Another $\log(n)$ idea

- What if we tried something more akin to an array.

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?
- Yes! For each i we can store the sum on the interval $[i, i + 2^j - 1]$ for $\log$ many j.

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?
- Yes! For each i we can store the sum on the interval $[i, i + 2^j - 1]$ for $\log$ many j.
- Then to retrieve a sum from i to j we always take the biggest chunk we can that's stored at i, which will always be at least half.

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?
- Yes! For each i we can store the sum on the interval $[i, i + 2^j - 1]$ for $\log$ many j.
- Then to retrieve a sum from i to j we always take the biggest chunk we can that's stored at i, which will always be at least half.
- Then we continue until we reach j, moving i along and collecting the results.

Another $\log(n)$ idea

- What if we tried something more akin to an array.
- Could we store $\log(n)$ amounts of data per element somehow?
- Yes! For each i we can store the sum on the interval $[i, i + 2^j - 1]$ for $\log$ many j.
- Then to retrieve a sum from i to j we always take the biggest chunk we can that's stored at i, which will always be at least half.
- Then we continue until we reach j, moving i along and collecting the results.
- This is what is known as a sparse table.

Sparse tables

- Calculating all of these values takes $\mathcal{O}(n \log(n))$ because we can calculate the values in order of increasing j.

Sparse tables

- Calculating all of these values takes $\mathcal{O}(n \log(n))$ because we can calculate the values in order of increasing j.
- Then when we calculate the sum of $[i, i + 2^j - 1]$ we just combine the earlier results of $[i, i + 2^{j-1} - 1]$ and $[i + 2^{j-1}, i + 2^j - 1]$.

Sparse tables

- Calculating all of these values takes $\mathcal{O}(n \log(n))$ because we can calculate the values in order of increasing j.
- Then when we calculate the sum of $[i, i + 2^j - 1]$ we just combine the earlier results of $[i, i + 2^{j-1} - 1]$ and $[i + 2^{j-1}, i + 2^j - 1]$.
- Querying takes $\mathcal{O}(\log(n))$, however updating is slow and difficult.

Sparse tables

- Calculating all of these values takes $\mathcal{O}(n\log(n))$ because we can calculate the values in order of increasing j.
- Then when we calculate the sum of $[i, i+2^j-1]$ we just combine the earlier results of $[i, i+2^{j-1}-1]$ and $[i+2^{j-1}, i+2^j-1]$.
- Querying takes $\mathcal{O}(\log(n))$, however updating is slow and difficult.
- Why would we then ever use this instead of segment trees?

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.
- Let us consider binary lifting in particular.

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.
- Let us consider binary lifting in particular.
- Suppose we have some function f that rearranges the values $\{0, 1, \ldots, n-1\}$ and we get q queries asking what happens to x if we apply f exactly m times to x.

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.
- Let us consider binary lifting in particular.
- Suppose we have some function f that rearranges the values $\{0, 1, \ldots, n-1\}$ and we get q queries asking what happens to x if we apply f exactly m times to x.
- The naïve solution is to calculate it every time, giving a time complexity of $\mathcal{O}(qm\mathcal{O}(f))$.

Binary lifting

- The reason might be is that with sparse tables we can do many things that segment trees can not because of how the results are combined.
- Let us consider binary lifting in particular.
- Suppose we have some function f that rearranges the values $\{0, 1, \ldots, n-1\}$ and we get q queries asking what happens to x if we apply f exactly m times to x.
- The naïve solution is to calculate it every time, giving a time complexity of $\mathcal{O}(qm\mathcal{O}(f))$.
- How might we use sparse tables to do better?

Binary lifting ctd.

- Let $f^{[y]}(x)$ denote the result of applying f exactly y times to x

Binary lifting ctd.

- Let $f^{[y]}(x)$ denote the result of applying f exactly y times to x
- For each i we store $f^{[2^j]}(i)$ as a sparse table

Binary lifting ctd.

- Let $f^{[y]}(x)$ denote the result of applying f exactly y times to x
- For each i we store $f^{[2^j]}(i)$ as a sparse table
- Then we can compute these in increasing order of j, calculating $j = 1$ using f itself and then for larger j letting $f^{[2^j]}(x) = f^{[2^{j-1}]}(f^{[2^{j-1}]}(x))$

Binary lifting ctd.

- Let $f^{[y]}(x)$ denote the result of applying f exactly y times to x
- For each i we store $f^{[2^j]}(i)$ as a sparse table
- Then we can compute these in increasing order of j, calculating $j = 1$ using f itself and then for larger j letting $f^{[2^j]}(x) = f^{[2^{j-1}]}(f^{[2^{j-1}]}(x))$
- Thus we can precompute the table in $\mathcal{O}(n(\mathcal{O}(f) + \log(n)))$ and each query takes $\mathcal{O}(\log(m))$, a much better time complexity

Sparse table example

7	1	6	4	8	0	9	2	2	7	1	6

$j = 0$

Sparse table example

												$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8												$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7											$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10										$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12									$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8								$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9							$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11						$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4					$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4	9				$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4	9	8			$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4	9	8	7		$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

8	7	10	12	8	9	11	4	9	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

18	19	18	21	19	13	20	12	16	14	7	6	$j = 2$
8	7	10	12	8	9	11	4	9	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

37	32	38	33	35	27	27	18	16	14	7	6	$j = 3$
18	19	18	21	19	13	20	12	16	14	7	6	$j = 2$
8	7	10	12	8	9	11	4	9	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Sparse table example

```
query(1, 8) = 19 + 9 + 2
```

37	32	38	33	35	27	27	18	16	14	7	6	$j = 3$
18	$\boxed{19}$	18	21	19	13	20	12	16	14	7	6	$j = 2$
8	7	10	12	8	$\boxed{9}$	11	4	9	8	7	6	$j = 1$
7	1	6	4	8	0	9	$\boxed{2}$	2	7	1	6	$j = 0$

Sparse table example

`query(0, 9) = 37 + 9`

$\boxed{37}$	32	38	33	35	27	27	18	16	14	7	6	$j = 3$
18	19	18	21	19	13	20	12	16	14	7	6	$j = 2$
8	7	10	12	8	9	11	4	$\boxed{9}$	8	7	6	$j = 1$
7	1	6	4	8	0	9	2	2	7	1	6	$j = 0$

Example problem: Stikl

- https://open.kattis.com/problems/stikl