# 1 Arrays with Super Powers

Today we're going to take a break from abstract high-powered algorithms and talk about some fairly low-level programming tricks.

Suppose we have a sequence of $n$ variables $a[0], a[1], \ldots, a[n-1]$, and we want to support the following operations on these variables:

| | |
|---|---|
| Assign$(i, x)$: | Execute the assignment $a[i] \leftarrow x$. |
| RangeSum$(i, j)$: | Return $\sum_{i \le k \le j} a[k]$. |

Furthermore we want the operations to run in $O(\log n)$ time and use $O(n)$ space.

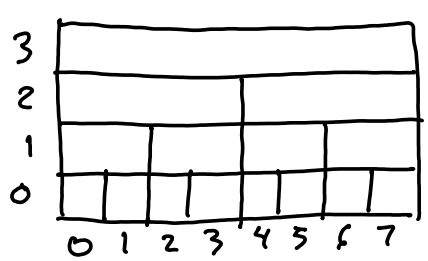
In this lecture we will present two alternative algorithms for solving this problem. We will discuss the details of how to write very short, elegant, and fast code for this. The two data structures are called SegTrees[1] and Fenwick trees, which are also known as binary index trees.

We'll also discuss how to generalize SegTrees (and to some extent Fenwick trees) to handle a much wider class of problems, where the $\sum$ operation is replaced by an arbitrary associative operator.

Note that these problems can be solved within the same time and space bounds by using augmented binary search trees. But the methods we present here are much simpler to implement and much faster in practice, although they are not as general. In particular binary search trees can better handle changing the length of the sequence of variables, splitting the sequence, and joining two sequences together.

# 2 SegTrees

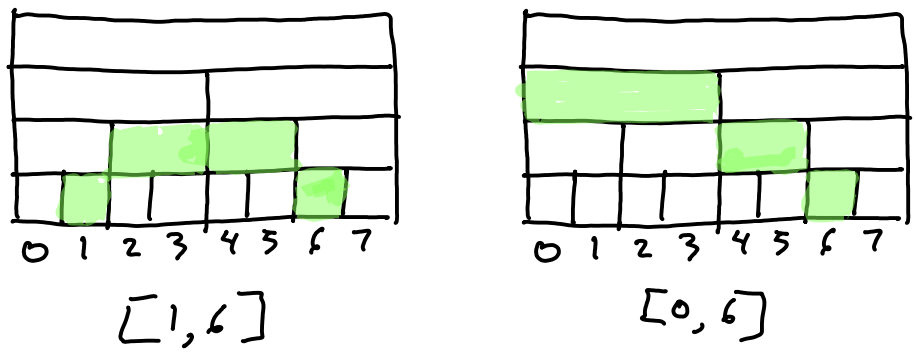
For the moment let's assume that $n$ is a power of 2. (If it is not a power of two, we can round it up to the next one. This is how the code at the end of the lecture works.) Consider the following diagram, which illustrates the case $n = 8$. The $i$th rectangle from the left in level 0 will store $a[i]$. Each box in each of the levels above will store the sum of all of the variables that are in that box if it were projected straight down to level 0. For example, the leftmost box in level 1 stores $a[0] + a[1]$. There are $\log_2 n$ levels.



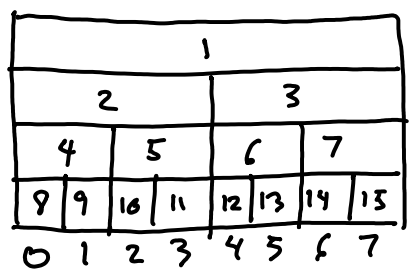

[1] "SegTree" is the name I have chosen to call this data structure. In the competition programming literature they are called "segment trees". This name conflicts with another specifically augmented binary search tree to represent a set of line segments in the plane. I have chosen the name "SegTree" to avoid any such ambiguity.

To execute an Assign$(i, x)$ we first put $x$ into the $i$th box on level 0. Then for each box intersecting the vertical line up from $i$ we recompute its value by adding together the values of the two boxes below it.

Now if we want to compute the sum of any range of these variables we can do it by selecting a subset of boxes to sum (at most two per level). Below on the left are the boxes we need to include to compute RangeSum$(1, 6)$, and on the right are the ones we need to compute RangeSum$(0, 6)$.




This data structure can be stored in a single array of length $2n$. We make use of the key trick of heap sort. Number the boxes starting at 1, in left to right order from the highest level down. You can see that the two boxes below box $i$ are $2i$ and $2i + 1$. Conversely the box above $i$ is $\lfloor i/2 \rfloor$. We've implicitly defined a tree with these operators being LeftChild$(i)$, RightChild$(i)$ and Parent$(i)$ respectively.




Let $2N$ be the size of the array ($N$ is 8 in the diagram above). Here is the `C` code for Assign$(i, x)$.

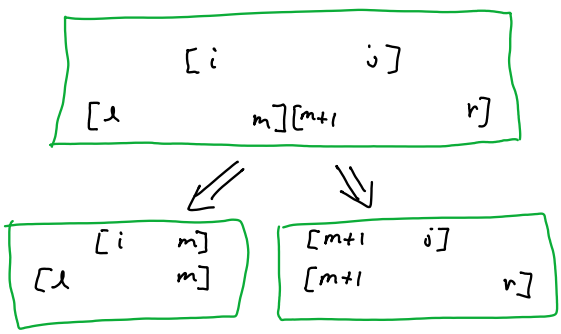
```
void Assign(int i, int x) {
    i = i+N;
    A[i] = x;
    for (i = Parent(i); i>0; i = Parent(i)) {
        A[i] = A[LeftChild(i)] + A[RightChild(i)];
    }
}
```

Note that for each box that changes we compute its value using the two boxes below it.

Now, there is a very elegant way in which to specify the set of boxes that will be summed in order to compute a RangeSum(). It's done by the following function `f()`.

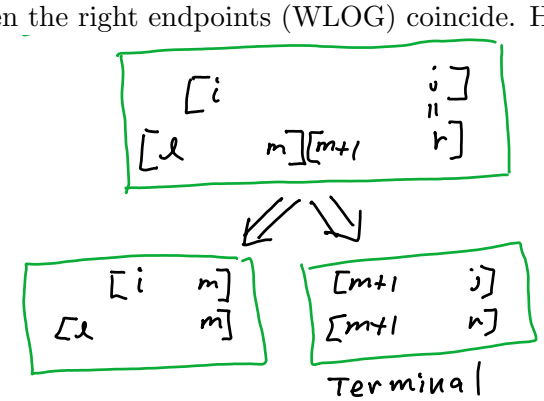
```
int f (int v, int l, int r, int i, int j) {
    /* We're currently at A[v].  1 <= v < 2*N.
       The range [l,r] is that of the current block, wrt user variables [0,n-1].
       The range [i,j] is the range of the query, wrt user variables [0,n-1].
       The size of the range [l,r] = r-l+1 is a power of 2.
       The range [l,r] contains the range [i,j].
       This function returns the sum the variables in the range [i,j].
     */
    int t1, t2, m;
    if (l==i && r==j) {
        return A[v];
    } else {
        m = (l+r)/2; /* split [l,r] into [l,m] [m+1,r] */
        t1 = (i <= m)? f (LeftChild(v), l, m, i, (min(m,j))): 0;
        t2 = (j >  m)? f (RightChild(v), (m+1), r, (max(i,(m+1))), j): 0;
        return t1 + t2;
    }
}
```

The following diagram shows how this works in the general case when none of the endpoints of the query interval $[i, j]$ coincide with the block $[l, r]$.




Note that the two recursive calls that it spins off have the property that one of the end points (at least) of the new query range and the new block coincide. So the general case shown above can only happen once. After this, in all the calls, at least one of these endpoints coincide.

Now consider the case when the right endpoints (WLOG) coincide. Here's what happens;




Note that this also spawns two recursive calls. But the one on the right is an immediate termial case. As a result, the recurive algorithm follows at most two paths down the tree. Thus the running

time is $O(\log n)$

A full implementation appears at the end of these notes.

### 2.1 Extensions of SegTrees

Note that in the implementation a function called `glue(a,b)` is used in the code in both places where we need to combine the values in two child boxes to compute the value in the parent box. This is because there's nothing special about addition. The entire system works with any associative operation. For example, we might want the maximum of the elements in the range instead of the sum. So we just replace `glue(a,b)` to compute the maximum of its arguments. The variable `identity` then needs to be set to the identity element of the glue operator.

Other kinds of generalizations are also possible. For example it's possible to add the capability of adding a constant $c$ to all variables in some range $[i, j]$. These kinds of additional capabilities are very useful.

## 3 Fenwick Trees (a.k.a. Binary Index Trees) (optional material)

Consider the binary represented integers $1, 2, \ldots n$. Let's define two functions on these: `UP` and `DN`. (Here `&` is the bitwise and operator.)

```
UP(i) = i + (i & -i)
DN(i) = i - (i & -i)
```

It's easy to see what these functions do when we look at the binary representation of `i` (we're assuming that `i` $> 0$). `i & -i` creates a word consisting of the rightmost 1 bit of i. So `DN(i)` just takes the rigthmost 1 bit of `i` and zeros it. `UP(i)` takes the rightmost 1 bit of `i`, zeros it, and carries a 1 to the left.

For example:

```
UP(01110000) = 10000000
DN(01110000) = 01100000
```

`UP()` is a strictly increasing function and `DN()` is strictly decreasing.

Let `UP-PATH(a)` be the sequence {`a, UP(a), UP(UP(a))`...} terminating before it exceeds $n$. Similarly let `DN-PATH(b)` be {`b,DN(b), DN(DN(b))` ...} terminating before it reaches 0.

**Theorem 1** *If* `a` $\leq$ `b` *then* `UP-PATH(a)` *and* `DN-PATH(b)` *intersect at precisely one point. Furthermore if* `a`>`b` *then these sets have no intersection.*

**Proof:** The 2nd part is clear because all elements of `UP-PATH(a)` are $\geq$ `a` and all elements of `DN-PATH(b)` are $\leq$ `b`. The result follows. Similarly if `a`=`b` then the two paths clearly intersect only at `a`. This leaves only the case `a`<`b` to consider.

First note that the `DN` operator applied to `b` just lops off one bit from the right. And the `UP` operator when applied to `a` finds a zero followed by the rightmost block of ones, and complements those bits.

The proof will be in two parts. We will show that there is some value `c` that is on both of the paths. Then we will show that there can be no other place where the paths intersect.

Consider the binary representations of `a` and `b`, for example.

```
a = 1010xxxxxx
b = 1011100101
```

Since a<b, in the leftmost position where they differ a has a 0 and b has a 1. Call that position $d$. If all the bits in a after $d$ (the x's above) are 0, then DN-PATH(b) will clearly reach a. Thus the intersection point is c=a.

If some of the xs above are 1 then let c be the number whose bits equal b's from the left end to $d$ and 0 afterwards. In this case

```
c = 1011000000
```

Clearly DN-PATH(b) is guaranteed to reach c. Similarly the UP operator applied to a will eventually reach c. (Recall that the bits after position $d$ in a are not all zero.) For example:

```
        a = 1010111010
     UP(a) = 1010111100
UP(UP(a)) = 1011000000 = c
```

Now all that remains is to prove that there can be no other intersection between these two paths. The number of trailing zeros in the sequence DN-PATH(b) is strictly increasing. (DN() lops off one bit each step.) Similarly the number of trailing zeros in the sequence UP-PATH(a) is also strictly increasing.

Let $k$ be the number of trailing zeros in c. Each element of DN-PATH(b) that is >c has $< k$ trailing zeros in it. Each element of UP-PATH(a) that is >c has $> k$ trailing zeros in it. Thus there can be no collision at a value z>c.

Similarly each element of DN-PATH(b) that is <c has $> k$ trailing zeros in it. Each element of UP-PATH(a) that is <c has $< k$ trailing zeros in it. Thus there can be no collision at a value z<c. ■

Why is this useful? Suppose we want a data structure that represents a sequence of numbers $x_1, x_2, \ldots, x_n$ (initially 0), with the following operations:

| | |
|---|---|
| Increment$(i, v)$: | $x_i \leftarrow x_i + v$ |
| SumToLeft$(j)$: | return $\sum_{1 \le i \le j} x_i$ |

To do this we keep an array A[1...n], initially zero. To do Increment$(i, v)$ we simply add $v$ to all the A[] values on UP-PATH(i).

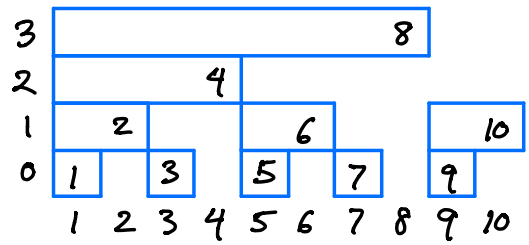
To compute SumToLeft$(j)$ we just return the sum of the A[] values on DN-PATH(j).

Both of these are $O(\log n)$.

This is guaranteed to work because of the theorem. Specifically the value computed by SumToLeft$(j)$ will include precisely one copy of all the increments that were applied to an $i \le j$, because these paths intersect once. And this path will not include any of the increments that were applied to any $i > j$.

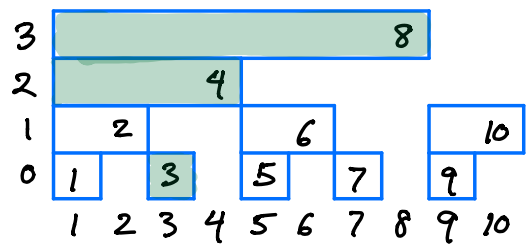
In the following diagram each horizontal position from 1 to 10 represents one of the variables. As in our SegTree diagrams, each box corresponds to one element of our array. The horizontal extent of the box shows which of the variables are included in it. So drawing a vertical line up from an

index hits the boxes on the `UP-PATH`. The `DN-PATH`s also move up levels, but also skip to the left, and include a set of boxes that contain all the indices from the starting index to 1.



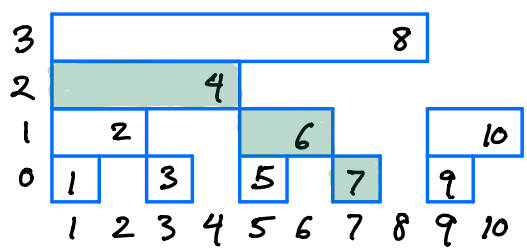

Each row of boxes is distinguished by the number of trailing zeros in the binary representation of its number.

In the following figure, the green boxes are those which are hit by `UP-PATH(3)`.




`UP-PATH(3)` = {`3,4,8`} = {`11, 100, 1000`}

In the following figure, the shaded boxes show `DN-PATH(7)`




`DN-PATH(7)` = {`7,6,4`} = {`111, 110, 100`}

An implementation in `C` is included below.

Note that this data structure does not support assigning a value to the variable $x_i$. This is because it does not actually store $x_i$ anywhere. (This could be remedied by adding a new array that stores the current values.) The Fenwick tree uses an array of size $n$, unlike the SegTree, which could use space up to $4n$.

Here are some additional observations. First of all it's possible to generalize this scheme to make shorter `UP-PATHS` and longer `DN-PATHS`.

For example by using base 4, we can make the `UP-PATHS` $\log_4(n)$ and the `DN-PATHS` $3\log_4(n)$. This kind of approach would be possibly useful if there were a lot more Increments than SumToLeft operations.

Another observation is that the theorem about these paths having a unique intersection when $a < b$ means that we can swap the role of `UP-PATHS` and `DN-PATHS`. So we could use `DN-PATH`s for the Increments and `UP-PATH`s for the sums. In this case the sum would be for all items from $i$ to $n$.

# 4 Implementation of SegTrees

```
#include <stdio.h>
#include <stdlib.h>

int SuperCeiling(int n) {/* The smallest power of 2 greater than or equal to n */
    int p;
    for (p=1; p < n; p = 2*p);
    return p;
}

int max (int a, int b) {return (a>b)? a: b;}
int min (int a, int b) {return (a<b)? a: b;}
int Parent(int i) {return i/2;}
int LeftChild(int i) {return 2*i;}
int RightChild(int i) {return 2*i+1;}

int glue(int a, int b) {  /* an arbitrary associative operator on elements */
    return a+b;
}
int identity = 0;         /* the identity element for the glue() operator */
int n, N, * A;

void Assign(int i, int x) {
    i = i+N;
    A[i] = x;
    for (i = Parent(i); i>0; i = Parent(i)) {
        A[i] = glue (A[LeftChild(i)], A[RightChild(i)]);
    }
}

int f (int v, int l, int r, int i, int j) {
    /* We're currently at A[v].  1 <= v < 2*N.
       The range [l,r] is that of the current block, wrt user variables [0,n-1].
       The range [i,j] is the range of the query, wrt user variables [0,n-1].
       The size of the range [l,r] = r-l+1 is a power of 2.
       The range [l,r] contains the range [i,j].
       This function returns the answer to the query.
     */
    int t1, t2, m;
    if (l==i && r==j) {
        return A[v];
    } else {
        m = (l+r)/2; /* split [l,r] into [l,m] [m+1,r] */
        t1 = (i <= m)? f (LeftChild(v), l, m, i, (min(m,j))): identity;
        t2 = (j >  m)? f (RightChild(v), (m+1), r, (max(i,(m+1))), j): identity;
        return glue (t1, t2);
    }
}

int RangeSum(int i, int j) {
    return f (1, 0, (N-1), i, j);
}
```

```
int main(){
    int i;
    n = 7;
    N = SuperCeiling(n);
    A = (int *) malloc (sizeof(int) * (2*N));
    for (i=0; i<2*N; i++) A[i] = identity;
    Assign(3,7);
    Assign(4,1);
    for (i=0; i<2*N; i++) {
        printf("A[%d] = %d\n", i, A[i]);
    }
    printf(" RangeSum(2,7) = %d\n", RangeSum(2,7));
    printf(" RangeSum(0,3) = %d\n", RangeSum(0,3));
    printf(" RangeSum(4,5) = %d\n", RangeSum(4,5));
    printf(" RangeSum(5,5) = %d\n", RangeSum(5,5));
}
```

# 5 Implementation of Fenwick Trees

```
#include <stdio.h>
#include <stdlib.h>

int n, * A;

void Increment(int i, int x) {
    for (;i <= n; i = i + (i & (-i))) {
        A[i] = A[i] + x;
    }
}

int SumToLeft(int i) { /* return A[1]+A[2]+...+A[i] */
    int t = 0;
    for (; i>0; i = i - (i & (-i))) {
        t += A[i];
    }
    return t;
}
```

```
int main(){
    int i;
    n = 11;
    A = (int *) malloc (sizeof(int) * (n+1));
    for (i=0; i <= n; i++) A[i] = 0;
    Increment(2,6);
    Increment(3,2);
    Increment(4,1);
    for (i=1; i<=n; i++) {
        printf("A[%d] = %d\n", i, A[i]);
    }

    for (i=1; i<6; i++) {
        printf(" SumToLeft(%d) = %d\n", i, SumToLeft(i));
    }
}
```