We shall discuss two algorithms today and we wiil try to improve the performance.

1. Lower bound, upper bound and equal range:

Let us say we have sorted the students based on their grades. We want to find all those who have got A grade. So, We want the leftmost member in the student array who has got A – that is called the lower bound. The rightmost member with A grade is upper bound. The elements between these to form the equal range.

Can we use binary search on a sorted array to find the lower bound or the upper bound?

Binary search returns the position of any element in that range. Then we may have move left to find the lower bound and we may have to move right to find the upper bound.

Check : problem1/ex1.c

// find lowerboud, upperbound and # of elements

// array in non decreasing order with repeating elements

#include <stdio.h>

int binary\_search(int a[], int n, int e)

{

int l = 0; int r = n - 1;

int m = (l + r) / 2; // assume no overflow!!

while(l <= r && a[m] != e)

{

if(a[m] > e)

{

r = m - 1;

}

else

{

l = m + 1;

}

m = (l + r) / 2;

}

return (l <= r) ? m : -1;

}

// call binary search

// if the search fails, return -1

// otherwise, move left as long as there is element in the array and the array element matches the

// given element.

// return the position of last match

int lower\_bound(int a[], int n, int e)

{

int m = binary\_search(a, n, e);

if(m == -1)

{

return -1;

}

else

{

while(m > 0 && a[m - 1] == e)

{

--m;

}

return m;

}

}

// similar to lower bound

// call binary search

// if it fails, return -1

// else

// move right as long as there is an element in the array and that element matches the given element

// return the position of last match

int upper\_bound(int a[], int n, int e)

{

int m = binary\_search(a, n, e);

if(m == -1)

{

return -1;

}

else

{

while(m < n - 1 && a[m + 1] == e)

{

++m;

}

return m;

}

}

void tester(int a[], int n, int e)

{

int lb = lower\_bound(a, n, e);

if(lb == -1)

{

printf("element : %d not found\n", e);

}

else

{

int ub = upper\_bound(a, n, e);

printf("element : %d ", e);

printf("lower bound : %d upper bound : %d "

" # of elem in range : %d\n", lb, ub, (ub - lb + 1));

}

}

int main()

{

int a[] = {2, 2, 2, 3, 5, 5, 5, 5, 7, 7};

int n = 10;

tester(a, n, 2);

tester(a, n, 3);

tester(a, n, 5);

tester(a, n, 7);

tester(a, n, 6);

}

Array:

int a[] = {2, 2, 2, 3, 5, 5, 5, 5, 7, 7};

Output:

$ gcc ex1.c

$ ./a.out

element : 2 lower bound : 0 upper bound : 2 # of elem in range : 3

element : 3 lower bound : 3 upper bound : 3 # of elem in range : 1

element : 5 lower bound : 4 upper bound : 7 # of elem in range : 4

element : 7 lower bound : 8 upper bound : 9 # of elem in range : 2

element : 6 not found

Can we optimize if both are found in the same routine?

Can we use sentinels to avoid bounds checking?

Even though we used binary search to location an element in the range, then finding lower bound and upper bound became linear. How do we use binary search to find lower bound and upper bound?

The functions are a bit deep. Please pay close attention to these functions in problem1/ex2.c

// find lowerboud, upperbound and # of elements

// array in non decreasing order with repeating elements

#include <stdio.h>

int lower\_bound(int a[], int n, int e)

{

int l = -1; int r = n;

int m = -1;

while(l + 1 != r)

{

m = (l + r) / 2;

if(a[m] >= e)

{

r = m;

}

else

{

l = m;

}

}

return (a[r] == e) ? r : -1;

}

int upper\_bound(int a[], int n, int e)

{

int l = -1; int r = n;

int m = -1;

while(l + 1 != r)

{

m = (l + r) / 2;

if(a[m] <= e)

{

l = m;

}

else

{

r = m;

}

}

return (a[l] == e) ? l : -1;

}

void tester(int a[], int n, int e)

{

int lb = lower\_bound(a, n, e);

if(lb == -1)

{

printf("element : %d not found\n", e);

}

else

{

int ub = upper\_bound(a, n, e);

printf("element : %d ", e);

printf("lower bound : %d upper bound : %d "

" # of elem in range : %d\n", lb, ub, (ub - lb + 1));

}

}

int main()

{

int a[] = {2, 2, 2, 3, 5, 5, 5, 5, 7, 7};

int n = 10;

tester(a, n, 2);

tester(a, n, 3);

tester(a, n, 5);

tester(a, n, 7);

tester(a, n, 6);

}

Output:

$ gcc ex2.c

$ ./a.out

element : 2 lower bound : 0 upper bound : 2 # of elem in range : 3

element : 3 lower bound : 3 upper bound : 3 # of elem in range : 1

element : 5 lower bound : 4 upper bound : 7 # of elem in range : 4

element : 7 lower bound : 8 upper bound : 9 # of elem in range : 2

element : 6 not found

Let us examine lower\_bound first.

Observe that l and r are initialized beyond the array. But we will never use them indices. So we do not require sentinels.

We will exit the while loop only when l + 1 == r. This implies that l and r are adjacent.

In the loop, we find the midpoint of the interval, change l and r based on the comparison a[m] >= e.

Unlike binary search, we do not make r equal to m – 1 or l equal m + 1.

We are not checking for a[m] == e explicitly.

If array is empty (n = 0), l + 1 == r in the very first testing. So loop is exited.

If array has one element (n = 1), then loop is entered, m becomes 0. Either l or r becomes m. In the next iteration, loop is exited.

Each time one of l or r will change. Both cannot become equal to each other as only one of them changes.

Ex: l = 5; r = 7; m = 6; either l = 6 or r = 6 then loop is exited

When the loop is exited, either a[r] = e or a[r] > e or a[r] < e(is the last one possible?).

If a[r] = e, then we have found the lower bound. Observe why it is not l.

int lower\_bound(int a[], int n, int e)

{

int l = -1; int r = n;

int m = -1;

while(l + 1 != r)

{

m = (l + r) / 2;

if(a[m] >= e)

{

r = m;

}

else

{

l = m;

}

}

return (a[r] == e) ? r : -1;

}

Similar idea l = m and r = m would not work in normal binary search. Why?

Let us observe the changes from lower\_bound to upper\_bound.

There are 3 changes.

1. condition checking is changed.

2. bodies of if and else swapped.

3. check for a[l] while returning.

int upper\_bound(int a[], int n, int e)

{

int l = -1; int r = n;

int m = -1;

while(l + 1 != r)

{

m = (l + r) / 2;

**if(a[m] <= e) // comparison changed**

{

l = m; **// bodies of if and else interchanged**

}

else

{

r = m;

}

}

return (a[l] == e) ? l : -1; **// check for a[l]**

}

Let us turn our attention to max sum of sub sequence problem.

Given an array of elements - positive and negative, find the maximum sum of a continuous sequence of elements. We want a positive sum.

If all the elements are positive, then the sum is that of the whole array.

If all the elements are negative, then we consider the sum as 0.

We will try a # of possible variations.

We will stored the result in a structure seqsum\_t.

The range of the continuous sequence is indicated by l and r, both inclusive.

The sum is stored in the field sum.

We will concentrate on the find\_sum.

There are 3 variables i, j, k.

The variable i : leftmost point of the sequence. This variable could vary from 0 to n – 1.

can we stop at n – 2?

The variable j : rightmost point of the sequence. This variable indicates the position i and anything to the right of it. So, j varies from i to n – 1.

The variable k varies from i to j, provides traversal across the subsequence to find the sum.

The current max is updated if the sum from pos i to pos j both inclusive exceeds current max.

Do try for large arrays and let me know the biggest array you could handle in reasonable time.

Check problem2/ex1.c.

//find the max sum of a contiguous subsequence

// brute force : cubic algorithm

#include <stdio.h>

struct seqsum

{

int l;

int r;

double sum;

};

typedef struct seqsum seqsum\_t;

seqsum\_t find\_sum(double a[], int n)

{

seqsum\_t s;

double current\_max = 0;

int current\_i = -1;

int current\_j = -1;

for(int i = 0; i < n; ++i)

{

for(int j = i; j < n; ++j)

{

double sum = 0;

for(int k = i; k <= j; ++k)

{

sum += a[k];

}

if(sum > current\_max)

{

current\_max = sum;

current\_i = i;

current\_j = j;

}

}

}

s.l = current\_i; s.r = current\_j; s.sum = current\_max;

return s;

}

int main()

{

double a[] = { 31, -41, 59, 26, -53, 58, 97, -93, -23, 84 };

int n = 10;

seqsum\_t s;

s = find\_sum(a, n);

printf("l : %d r : %d sum : %lf\n", s.l, s.r, s.sum);

}

output:

$ gcc ex1.c

$ ./a.out

l : 2 r : 6 sum : 187.000000

Our first attempt is cubic. Can we make this better?

Check this code from problem2/ex2.c

Observe that for a particular value of i, summation from a[i] to a[j] is same as sum from a[i] to a[j-1] + a[j].

a[i .. j] = a[i .. j-1] + a[j].

So, we can knock off the k-loop.

typedef struct seqsum seqsum\_t;

seqsum\_t find\_sum(double a[], int n)

{

seqsum\_t s;

double current\_max = 0;

int current\_i = -1;

int current\_j = -1;

for(int i = 0; i < n; ++i)

{

**double sum = 0;**

for(int j = i; j < n; ++j)

{

**sum += a[j]; // sum from a[i] to a[j]**

if(sum > current\_max)

{

current\_max = sum;

current\_i = i;

current\_j = j;

}

}

}

s.l = current\_i; s.r = current\_j; s.sum = current\_max;

return s;

}

Output:

$ gcc ex2.c

$ ./a.out

l : 2 r : 6 sum : 187.000000

now the code is quadratic.

Can you try to find the biggest array for which this algorithm works in reasonable time?

There is one other way, we could make the code quadratic.

Check problem2/ex3.c.

Store cumulative sums in an array – acc in this case.

The variable acc[0] is 0.

The entry acc[i] is sum of elements from a[0] to a[i-1]

to find sum of a[i] to a[j] : s**um = acc[j-1] – acc[i-1];**

Observe how current\_i and current\_j are updated.

seqsum\_t find\_sum(double a[], int n)

{

seqsum\_t s;

double current\_max = 0;

int current\_i = -1;

int current\_j = -1;

**double acc[n + 1];**

acc[0] = 0;

**for(int i = 1; i <= n; ++i)**

**{**

**acc[i] = acc[i-1] + a[i-1];**

**}**

for(int i = 1; i <= n; ++i)

{

double sum = 0;

for(int j = i; j < n; ++j)

{

s**um = acc[j] - acc[i-1];**

if(sum > current\_max)

{

current\_max = sum;

**current\_i = i - 1;**

**current\_j = j - 1;**

}

}

}

s.l = current\_i; s.r = current\_j; s.sum = current\_max;

return s;

}

Output:$ gcc ex3.c

$ ./a.out

l : 2 r : 6 sum : 187.000000

Can we use divide and conquer and make the function O(n log n)?

check : problem2/ex4.c

**This function compares 3 structures a, b, c and finds which of these has higher value of sum and returns that structure as the result**

seqsum\_t max3(seqsum\_t a, seqsum\_t b, seqsum\_t c)

{

if(a.sum > b.sum && a.sum > c.sum)

{

return a;

}

else if(b.sum > c.sum)

{

return b;

}

else

{

return c;

}

}

Use divide and conquer as in mergesort.

Divide into two equal halfs.

Find the solution in the left half. Find the solution in the right half.

Find whether the solution can exist across the mid point.

Find the max of there three.

That is the result.

Check the base cases.

Array is empty. Result is 0.

seqsum\_t s = {-1, -1, 0.0};

Array has one element. If it positive, then that is the sum. Otherwise sum is 0.

seqsum\_t s = {l, r, (a[l] > 0) ? a[l] : 0};

This code finds the max in the left portion of mid point.

lmax = sum = 0.0;

for(int i = m; i >= l;--i)

{

sum += a[i];

if(sum > lmax)

{

lmax = sum;

current\_i = i;

}

}

This code finds the max in the right portion of mid point.

rmax = sum = 0.0;

for(int i = m + 1; i <= r; ++i)

{

sum += a[i];

if(sum > rmax)

{

rmax = sum;

current\_j = i;

}

}

So, lmax + rmax is the sum across the midpoint.

Find max in left half, find max in right half, find max across the midpont.

Amongst these, find the highest.

seqsum\_t s\_l\_r = {current\_i, current\_j, lmax + rmax};

seqsum\_t s\_l = find\_sum\_r(a, l, m);

seqsum\_t s\_r = find\_sum\_r(a, m + 1, r);

return max3(s\_l\_r, s\_l, s\_r);

seqsum\_t find\_sum\_r(double a[], int l, int r)

{

if(l > r)

{

seqsum\_t s = {-1, -1, 0.0};

return s;

}

if(l == r)

{

seqsum\_t s = {l, r, (a[l] > 0) ? a[l] : 0};

return s;

}

int m = (l + r) / 2;

int current\_i = l;

int current\_j = m + 1;

double lmax; double rmax; double sum;

lmax = sum = 0.0;

for(int i = m; i >= l;--i)

{

sum += a[i];

if(sum > lmax)

{

lmax = sum;

current\_i = i;

}

}

rmax = sum = 0.0;

for(int i = m + 1; i <= r; ++i)

{

sum += a[i];

if(sum > rmax)

{

rmax = sum;

current\_j = i;

}

}

seqsum\_t s\_l\_r = {current\_i, current\_j, lmax + rmax};

seqsum\_t s\_l = find\_sum\_r(a, l, m);

seqsum\_t s\_r = find\_sum\_r(a, m + 1, r);

return max3(s\_l\_r, s\_l, s\_r);

}

Output:

$ gcc ex4.c

$ !.

./a.out

l : 2 r : 6 sum : 187.000000

Recurrence : T(n) = 2 T(n/2) + O(n)

Solution : O(n \* logn)

Can we make the algorithm linear?

Check ex5.c.

Observe when we stop adding to the sum in a subsequence. We do not stop if we get a negative number. This number could be followed by bigger numbers.

We do break this summation if the sum becomes negarive.

So, we use a variable max\_end to track the sum for the current sequence. We reset it to 0 when the sum becomes negative.

Now, the algorithm is linear.

seqsum\_t find\_sum(double a[], int n)

{

seqsum\_t s;

double current\_max = 0;

int current\_i = -1;

int current\_j = -1;

int max\_l = -1;

int max\_r = -1;

int max\_end = 0;

for(int i = 0; i < n; ++i)

{

if(max\_end + a[i] > 0)

{

max\_end += a[i];

max\_r = i;

}

else

{

max\_end = 0;

max\_l = i + 1;

}

if(max\_end > current\_max)

{

current\_max = max\_end;

current\_i = max\_l;

current\_j = max\_r;

}

}

s.l = current\_i; s.r = current\_j; s.sum = current\_max;

return s;

}

Output:

$ gcc ex5.c

$ ./a.out

l : 2 r : 6 sum : 187.000000

We shall stop at this. Experiment. Have a great time.