

Analysis of Algorithms

Dilemma: you have two (or more) methods to solve problem, how to choose the BEST?

One approach: implement each algorithm in C, test how long each takes to run.

Problems:

- Different implementations may cause an algorithm to run faster/slower
- Some algorithms run faster on some computers
- Algorithms may perform differently depending on data (e.g., sorting often depends on what is being sorted)

Outline

Analysis

Concept of "best"

What to measure

Types of "best"

best-, average-, worst-case

Comparison methods

Big-O analysis

Examples

Searching, sorted array: sequential vs. binary

Sorting: selection, insertion, merge

Analysis: A Better Approach

Idea: characterize performance in terms of key operation(s)

- Sorting:
 - count number of times two values compared
 - count number of times two values swapped
- Search:
 - count number of times value being searched for is compared to values in array
- Recursive function:
 - count number of recursive calls

Analysis in General

Want to comment on the "general" performance of the algorithm

- Measure for several examples, but what does this tell us in general?

Instead, assess performance in an abstract manner

Idea: analyze performance as size of problem grows

Examples:

- Sorting: how many comparisons for array of size N?
- Searching: #comparisons for array of size N

May be difficult to discover a reasonable formula

Analysis where Results Vary

Example: for some sorting algorithms, a sorting routine may require as few as $N-1$ comparisons and as many as $\frac{N^2}{2}$

Types of analyses:

- Best-case: what is the fastest an algorithm can run for a problem of size N?
- Average-case: on average how fast does an algorithm run for a problem of size N?
- Worst-case: what is the longest an algorithm can run for a problem of size N?

Computer scientists mostly use worst-case analysis

How to Compare Formulas?

Which is better: $50N^2 + 31N^3 + 24N + 15$ or $3N^2 + N + 21 + 4 \cdot 3^N$

Answer depends on value of N:

N	$50N^2 + 31N^3 + 24N + 15$	$3N^2 + N + 21 + 4 \cdot 3^N$
1	120	37
2	511	71
3	1374	159
4	2895	397
5	5260	1073
6	8655	3051
7	13266	8923
8	19279	26465
9	26880	79005
10	36255	236527

What Happened?

N	$3N^2 + N + 21 + 4 \cdot 3^N$	$4 \cdot 3^N$	%ofTotal
1	37	12	32.4
2	71	36	50.7
3	159	108	67.9
4	397	324	81.6
5	1073	972	90.6
6	3051	2916	95.6
7	8923	8748	98.0
8	26465	26244	99.2
9	79005	78732	99.7
10	236527	236196	99.9

– One term dominated the sum

As N Grows, Some Terms Dominate

Function	10	100	1000	10000	100000
$\log_2 N$	3	6	9	13	16
N	10	100	1000	10000	100000
$N \log_2 N$	30	664	9965	10^5	10^6
N^2	10^2	10^4	10^6	10^8	10^{10}
N^3	10^3	10^6	10^9	10^{12}	10^{15}
2^N	10^3	10^{30}	10^{301}	10^{3010}	10^{30103}

Order of Magnitude Analysis

Measure speed with respect to the part of the sum that grows quickest

$$50N^2 + \boxed{31N^3} + 24N + 15$$

$$3N^2 + N + 21 + \boxed{4 \cdot 3^N}$$

Ordering:

$$1 < \log_2 N < N < N \log_2 N < N^2 < N^3 < 2^N < 3^N$$

Order of Magnitude Analysis (cont)

Furthermore, simply ignore any constants in front of term and simply report general class of the term:

$$31 \boxed{N^3} \quad 4 \cdot \boxed{3^N} \quad 15 \boxed{N \log_2 N}$$

$50N^2 + 31N^3 + 24N + 15$ grows proportionally to N^3

$3N^2 + N + 21 + 4 \cdot 3^N$ grows proportionally to 3^N

When comparing algorithms, determine formula(s) to count operation(s) of interest, then compare dominant terms of formulas

Big O Notation

Algorithm A requires time proportional to $f(N)$ - algorithm is said to be of order $f(N)$ or $O(f(N))$

Definition: an algorithm is said to take time proportional to $O(f(N))$ if there is some constant C such that for all but a finite number of values of N, the time taken by the algorithm is less than $C \cdot f(N)$

Examples:

$$50N^2 + 31N^3 + 24N + 15 \text{ is } O(N^3)$$

$$3N^2 + N + 21 + 4 \cdot 3^N \text{ is } O(3^N)$$

If an algorithm is $O(f(N))$, $f(N)$ is said to be the *growth-rate* function of the algorithm

Example: Searching Sorted Array

Algorithm 1: Sequential Search

```
int search(int A[], int N, int Num) {
    int index = 0;

    while ((index < N) && (A[index] < Num))
        index++;

    if ((index < N) && (A[index] == Num))
        return index;
    else
        return -1;
}
```

Analyzing Search Algorithm 1

Operations to count: how many times Num is compared to member of array

One after the loop each time plus ...

Best-case: find the number we are looking for at the first position in the array (1 + 1 = 2 comparisons) $O(1)$

Average-case: find the number on average half-way down the array (sometimes longer, sometimes shorter)

($N/2+1$ comparisons) $O(N)$

Worst-case: have to compare Num to every element in the array ($N + 1$ comparisons) $O(N)$

Search Algorithm 2: Binary Search

```
int search(int A[], int N, int Num) {
    int first = 0;
    int last = N - 1;
    int mid = (first + last) / 2;
    while ((A[mid] != Num) && (first <= last)) {
        if (A[mid] > Num)
            last = mid - 1;
        else
            first = mid + 1;
        mid = (first + last) / 2;
    }
    if (A[mid] == Num)
        return mid;
    else
        return -1;
}
```

Analyzing Binary Search

One comparison after loop

First time through loop, toss half of array (2 comps)

Second time, half remainder (1/4 original) 2 comps

Third time, half remainder (1/8 original) 2 comps

...

Loop Iteration	Remaining Elements
1	$N/2$
2	$N/4$
3	$N/8$
4	$N/16$

...

?? 1

How long to get to 1?

Analyzing Binary Search (cont)

Looking at the problem in reverse, how long to double the number 1 until we get to N ?

$N = 2^X$ and solve for X

$$\log_2 N = \log_2(2^X) = X$$

two comparisons for each iteration, plus one comparison at the end -- binary search takes

$2\log_2 N + 1$ in the worst case

Binary search is worst-case $O(\log_2 N)$

Sequential search is worst-case $O(N)$

Analyzing Sorting Algorithms

Algorithm 1: Selection Sort

```
void sort(int A[], int N) {
    int J, K, SmallAt, Temp;
    for (J = 0; J < N-1; J++) {
        SmallAt = J;
        for (K = J+1; K < N; K++)
            if (A[K] < A[SmallAt])
                SmallAt = K;
        Temp = A[J];
        A[J] = A[SmallAt];
        A[SmallAt] = Temp;
    }
}
```

Sorting Operations of Interest

- Number of times elements in array compared
- Number of times element copied (moved)

Selection sort

J=0: set min as 0, compare min to each value in $A[1..N-1]$
swap (3 copies)

J=1: set min as 1, compare min to each value in $A[2..N-1]$
swap (3 copies)

J=2: set min as 2, compare min to each value in $A[3..N-1]$
swap (3 copies)

...

Analyzing Selection Sort

Comparisons:

$$(N-1) + (N-2) + (N-3) + \dots + 1 = \frac{(N-1)*N}{2}$$

$$O(N^2)$$

Copies: (for this version)

$$3*(N-1)$$

$$O(N)$$

Insertion Sort

```
void sort(int A[], int N) {
    int J, K, temp;
    for (J = 1; J < N; J++) {
        temp = A[J];
        for (K = J; (K > 0) && (A[K-1] > temp); K--)
            A[K] = A[K-1];
        A[K] = temp;
    }
}
```

Both compares and copies done during inner loop

Insertion Sort Analysis

J=1 may have to shift val at A[0] to right (at most 1 comp)
at most three copies (one to pick up val at position 1, one
to shift val at 0 to 1, one to put val down)

J=2 may have to shift vals at A[0..1] to right (at most 2 comps)
at most four copies (pick up val at A[2], shift A[1] to A[2],
A[0] to A[1], put val from A[2] down at A[0])

J=3 may have to shift vals at A[0..2] to right (at most 3 comps)
at most five copies (pick up val at A[3], shift A[2] to A[3],
A[1] to A[2], A[0] to A[1], put val from A[3] at A[0])

...

Insertion Sort Analysis (cont)

Comparisons (worst case):

$$1 + 2 + 3 + \dots + (N-1) = \frac{(N-1)*N}{2}$$

$$O(N^2)$$

Copies: (for this version)

$$(2+1) + (2+2) + \dots + (2+(N-1)) = 2(N-1) + \frac{(N-1)*N}{2}$$

$$O(N^2)$$

Insertion Sort - Best Case

Comparisons (best case):

$$1 + 1 + 1 + \dots + 1 = (N-1)$$

$$O(N)$$

Copies:

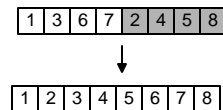
$$2(N-1)$$

$$O(N)$$

When? Sorted array (still $O(N)$ when *almost* sorted)

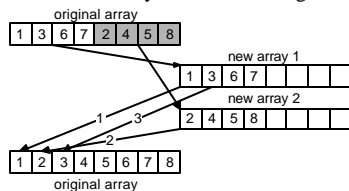
Merge Sort

Idea: divide the array in half, sort the two halves
(somehow), then “merge” the two sorted halves
into a complete sorted array



How to Merge

- Copy the two segments into two other arrays
- Copy the smaller of the two elements from the front of the two arrays back to the original array



Merging Two Array Segments

```
void merge(int A[], int s1low, int s1high,
           int s2low, int s2high) {
    int n1 = s1high - s1low + 1;
    int n2 = s2high - s2low + 1;
    int templ[ASIZE];
    int temp2[ASIZE];
    int dst, src1, src2;

    /* Copy A[s1low..s1high] to templ[0..n1-1] */
    copy_to_array(A,templ,s1low,s1high,0);
    /* Copy A[s2low..s2high] to temp2[0..n2-1] */
    copy_to_array(A,temp2,s2low,s2high,0);
```

Copying Array Segment

```
void copy_to_array(int srcarray[], int dstarray[],
                  int slow, int shigh, int dst) {
    int src;

    /* Copy elements from srcarray[slow..shigh] to
       dstarray starting at position dst */
    for (src = slow; src <= shigh; src++) {
        dstarray[dst] = srcarray[src];
        dst++;
    }
}
```

Merge (continued)

```
dst = segllo; /* Move elements to A starting at
              position segllo */
src1 = 0; src2 = 0;
/* While there are elements left in templ, temp2 */
while ((src1 < n1) && (src2 < n2)) {
    /* Move the smallest to A */
    if (templ[src1] < temp2[src2]) {
        A[dst] = templ[src1];
        src1++;
    }
    else {
        A[dst] = temp2[src2];
        src2++;
    }
    dst++;
}
```

Merge (continued)

```
/* Once there are no elements left in either
   templ or temp2, move the remaining elements
   in the non-empty segment back to A */
if (src1 < n1)
    copy_to_array(templ,A,src1,n1-1,dst);
else
    copy_to_array(temp2,A,src2,n2-1,dst);
}
```

Merge Sorting

To merge, we need two halves of array to be sorted, how to achieve this:

- recursively call merge sort on each half
- base case for recursion: segment of array is so small it is already sorted (has 0 or 1 elements)
- need to call merge sort with segment to be sorted (0 to N-1)

```
void sort(int A[], int N) {
    do_merge_sort(A,0,N-1);
}
```

Merge Sort Recursive Function

```
void do_merge_sort(int A[], int low, int high) {
    int mid;
    /* Base case low >= high, we simply do not do
       anything, no need to sort */
    if (low < high) {
        mid = (low + high) / 2;
        do_merge_sort(A, low, mid);
        do_merge_sort(A, mid+1, high);
        merge(A, low, mid, mid+1, high);
    }
}
```

Merge Sort Analysis

Comparisons to merge two partitions of size X is $2X$ at most (each comparison causes us to put one element in place)

Copies is $4X$ (similar reasoning, but we have to move elements to temp arrays and back)

Partitions of Size	# Partitions	#Comparisons
$N/2$	2	N
$N/4$	4	N
$N/8$	8	N
...		
1	N	N

Merge Sort Analysis (cont)

Comparison cost is N * how many different partition sizes

of partition sizes related to how long to go from 1 to N by doubling each time ($\log_2 N$)

Cost: Comparisons $O(N \log_2 N)$

Copies $O(N \log_2 N)$