CSE 1030

Yves Lespérance

Lecture Notes

Week 10 — Algorithm Analysis, Searching and Sorting

Recommended Readings:
Van Breugel & Roumani Ch. 8 and Savitch Ch. 6

Program Efficiency

The cost of executing a program or algorithm can be measured by the amounts of *time* and *space* it uses. Generally, time is more critical.

Why should we want to know?

- to estimate the program's running time,
- to see how large an input it can cope with,
- to compare different algorithms for solving the same problem.

Can study efficiency by measuring actual running time on a particular computer for sample inputs of different sizes. Informative, but the results depends on hardware, language, and compiler used. Also time consuming.

Algorithm Analysis

Also possible to analyze efficiency of algorithms using mathematical methods; studied in area of CS called *analysis of algorithms*.

In general, execution time increases with the size of the input; e.g. computing factorial of n, sorting an array of size n. So use a function $T(n)$ of input size n to represent running time of algorithm.

However, running time may not be the same for all inputs of the same size, e.g. sorting. Can analyze *average case* running time, but this requires statistical methods. More often, analyze *worst case* running time. This is simpler, provides a guarantee, and often gives the same results.

We do the analysis by counting the number of operations performed by the algorithm for the worst input of size n.

Simple e.g., calculating $n! = n * (n-1) * \ldots * 1$:

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

$$\begin{aligned} T(n) &= n \text{ multiplications} + \\ &\quad n \text{ assignments} + \\ &\quad n \text{ tests} + \\ &\quad n \text{ incrementations} + \\ &\quad 2 \text{ assignments for initialization} + \\ &\quad 1 \text{ final test} \\ &= 4n + 3 \text{ operations} \end{aligned}$$

Approximating, we drop the lower order term 3 as it becomes negligeable when n gets large. Also, we drop the constant 4 since the time to perform an operation depends on the computer. So we say that $T(n)$ is $O(n)$, i.e. *Big-Oh* n.

Big-oh can be defined formally:

$f(n)$ is $O(g(n))$ iff there exist positive constants C and K such that $f(n) \leq C \cdot g(n)$ for all $n \geq K$.

Big-oh gives us an estimate of how fast running time grows as n grows.

In doing the analysis, don't need to count all operations, only pick one that reflects the overall performance of the algorithm — is executed most often, is in the inner most loop. E.g. for iterative factorial, count multiplications. E.g. for sorting: count array accesses or comparisons between elements.

Sorting

Sorting an array or list is a very common operation. The array/list is sorted if all its elements appear in the right order.

What "right order" means is application dependent. E.g. sort array of Student objects so that they appear in increasing order of student number. Could also sort by names using lexicographic order. Another e.g.: could sort marks in a class in decreasing order.

It is much easier to find an item if the array/list is sorted than if it is not (e.g. in a phone book).

There are many different algorithms for array sorting. Much work has been done to analyse them and determine which are the best in terms of running time.

Merge Sort — A Recursive Array Sorting Algorithm

Steps:

1. divide the array into 2 halves;
2. sort each half recursively;
3. merge the sorted halves back into a single array.

```
public static void mergeSort(int[] a, int from, int to)
{  if(from == to) return;
   int mid = (from + to) / 2;
   // sort both halves recursively and merge back
   mergeSort(a, from, mid);
   mergeSort(a, mid + 1, to);
   merge(a, from, mid, to);
}

public static void sort(int[] a)
{  mergeSort(a, 0, a.length - 1);
}
```



```
public static void merge(int[] a, int from, int mid, int to)
{  // create temporary array
   int both = to - from + 1;
   int[] tempA = new int[both];
   // merge until one array runs out
   int i1 = from;
   int i2 = mid + 1;
   int j = 0;
   while(i1 <= mid && i2 <= to)
   {  if (a[i1] < a[i2])
      {  tempA[j] = a[i1];
         i1++;
      }
      else
      {  tempA[j] = a[i2];
         i2++;
      }
      j++;
   }
```

```
   // copy rest of remaining half
   while(i1 <= mid){
      tempA[j] = a[i1];
      i1++;
      j++;
   }
   while(i2 <= to){
      tempA[j] = a[i2];
      i2++;
      j++;
   }
   // copy tempA back into a
   for (j = 0; j < both; j++)
      a[from + j] = tempA[j];
}
```


Tracing execution of Merge Sort on `a = [ 18, 33, 4, 21, 17, 35, 20 ]`

```
mergeSort(a,0, 6) calls mergeSort(a,0,3)
  mergeSort(a,0,3) calls mergeSort(a,0,1)
    mergeSort(a,0,1) calls mergeSort(a,0,0)
      mergeSort(a,0,0) returns
    mergeSort(a,0,1) calls mergeSort(a,1,1)
      mergeSort(a,1,1) returns
    mergeSort(a,0,1) calls merge(a,0,0,1)
      merge(a,0,0,1) merges [18] and [33] returns with a unchanged
    mergeSort(a,0,1) returns with a unchanged
  mergeSort(a,0,3) calls mergeSort(a,2,3)
    mergeSort(a,2,3) calls mergeSort(a,2,2)
      mergeSort(a,2,2) returns
    mergeSort(a,2,3) calls mergeSort(a,3,3)
      mergeSort(a,3,3) returns
    mergeSort(a,2,3) calls merge(a,2,2,3)
      merge(a,2,2,3) merges [4] and [21] returns with a unchanged
    mergeSort(a,2,3) returns with a unchanged
  mergeSort(a,0,3) calls merge(a,0,1,3)
    merge(a,0,1,3) merges [18, 33] and [4, 21]
      returns with a = [4, 18, 21, 33, 17, 35, 20]
  mergeSort(a,0,3) returns with a as above
```



```
 mergeSort(a,0,6) calls mergeSort(a,4,6)
  mergeSort(a,4,6) calls mergeSort(a,4,5)
   mergeSort(a,4,5) calls mergeSort(a,4,4)
      mergeSort(a,4,4) returns
    mergeSort(a,4,5) calls mergeSort(a,5,5)
      mergeSort(a,5,5) returns
    mergeSort(a,4,5) calls merge(a,4,4,5)
      merge(a,4,4,5) merges [17] and [35] returns with a unchanged
    mergeSort(a,4,5) returns with a unchanged
  mergeSort(a,4,6) calls mergeSort(a,6,6)
    mergeSort(a,6,6) returns
  mergeSort(a,4,6) calls merge(a,4,5,6)
    merge(a,4,5,5) merges [17, 35] and [20]
      returns with a = [4, 18, 21, 33, 17, 20, 35]
  mergeSort(a,4,6) returns with a as above
mergeSort(a,0,6) calls merge(a,0,3,6)
  merge(a,0,3,6) merges [4, 18, 21, 33] and [17, 20, 35]
    returns with a = [4, 17, 18, 20, 21, 33, 35]
mergeSort(a,0,6) returns with a as above
```


Proof by Induction of Correctness and Termination for Merge Sort

Let's assume that `merge(a,i,m,j)` is correct and terminates, i.e. merges sorted sub-arrays `a[i..m]` and `a[m+1..j]` into a single sorted sub-array `a[i..j]`. This can be proven using loop invariants.

We prove correctness and termination of the `mergeSort(a,i,j)` method by induction on the size n of the sub-array considered, i.e. $n = \mathtt{j} - (\mathtt{i} - 1)$.

1) Base case $n = 1$: then `mergeSort(a,i,j)` immediately returns, and a subarray of size 1 is always sorted. So the method is correct.

2) For the recursive case: Assume that `mergeSort(a,i,j)` is correct and terminates for all sizes $n \leq k$ (induction hypothesis).

We must prove that that the method is correct for size $n = k + 1$.

Then `mergeSort(a,i,j)` calls `mergeSort(a,i,mid)` and `mergeSort(a,mid+1,j)`. In both cases, the size of the sub-array involved is $\leq k$, so by the induction hypothesis these two calls correctly sort the sub-arrays `a[i..mid]` and `a[mid+1..j]`.

After the recursive call, `merge(a,i,mid,j)` is called to merge the two sorted sub-arrays. By our assumption that `merge` is correct, this results in `a[i,j]` being sorted.

So `mergeSort` is correct for size $n = k + 1$.

Thus for all natural numbers n `mergeSort(a,i,j)` correctly sorts the sub-array `a[i..j]` where n is the size of the sub-array and terminates.

Analysis of Running Time of Merge Sort

Merging 2 arrays containing a total of n elements requires $n - 1$ comparisons in the worst case (when neither half runs out early). If you want to count all operations, then the running time will be $C_1 n + C_2$ where C_1 and C_2 are constants. If you drop the lower order term, you get $C_1 n$.

The running time for the complete sorting of an array of size n can be specified as a *recurrence relation:*

$$T(n) = \begin{cases} C_2 & \text{if } n \leq 1 \\ 2T(n/2) + C_1 n & \text{otherwise} \end{cases}$$

There are general methods for solving recurrence relations, but let's do it from first principles. If $n/2 > 1$, we can apply the recurrence relation to $T(n/2)$ to get:

$$\begin{aligned} T(n) &= 2(2T(n/4) + C_1 n/2) + C_1 n \\ &= 4T(n/4) + 2C_1 n \end{aligned}$$

Suppose that n is a power of 2, i.e. $n = 2^m$. Then we can keep applying the relation until n reaches $2^0 = 1$:

$$\begin{aligned} T(n) &= 2^m T(n/2^m) + mC_1 n \\ &= C_2 2^m + C_1 mn \text{ since } T(n/2^m) = T(1) = C_2 \\ &= C_2 n + C_1 n \log_2 n \text{ since } m = \log_2 n \\ &= O(n \log n) \end{aligned}$$

Thus, the running time of merge sort grows much more slowly than that of many other sorting algorithms whose running time is $O(n^2)$, e.g. selection sort, insertion sort, etc.:

n	$n \log_e n$	n^2
10	23	100
100	460	$10,000$
10^3	$6.9\ 10^3$	10^6
10^6	$13.8\ 10^6$	10^{12}

Searching

We have seen that a common operation on arrays and collections is sorting them.

Another common operation is *searching* an array to locate a given value: you are given a value, the target, and you must return the index where it appears in the array; if it doesn't appear, you return some value to indicate that it was not there, such as a value like -1 that is not a valid index.

Like for sorting, there are many algorithms to perform searching. Will look at some and do an analysis.

Linear Search

One algorithm for searching an array is simply to start at the beginning and go through each element in sequence; for each element, you compare it with the value you are looking for, the target; you are done when you find the target or you reach the end of the array. This is called *linear search*.

```
public static int linearSearch(int[] a,
   int target)
{  for (int i = 0; i < a.length; i++)
      if (target == array[i])
         return i;
   return -1;
}
```


If you perform linear search on an array of size n, in the worst case you will have to compare the target with all of the array elements, i.e., do n comparisons; on average, you will compare the target with half of the array elements, i.e., do $n/2$ comparisons.

Thus, we say that in the worst case, linear search takes $O(n)$ operations.

Binary Search

When the array you are searching is already *sorted*, then there is a much more efficient algorithm.

You start by comparing the `target` with the element at the middle of the array. If `target == a[mid]`, you just return `mid` and you are done.

Otherwise, there are two cases:

- `target < a[mid]`, in which case `target` can only be between index `0` and `mid - 1`.
- `target > a[mid]`, in which case `target` can only be between index `mid + 1` and `lenght - 1`.

In either case, you've eliminated half of the array; you can continue by applying the same method to the remaining part of the array.

This method is called *binary search*.

Here is an iterative implementation:

```
public static int binarySearch(int[] a,
   int target)
{
   int from = 0;
   int to = a.length - 1;
   while (from <= to)
   {  int mid = (from + to) / 2;
      if (a[mid] == target)
         return mid;
      else if (target < a[mid])
         to = mid - 1;
      else
         from = mid + 1;
   }
   return -1;
}
```

See textbook for a recursive one.

Analysis

Suppose we have an array of size n. In the worst case, we have that:

# of comparisons	# of elements remaining
0	n
1	$n/2$
2	$n/4$
...	
$\log_2 n$	1

So binary search requires in the order of $\log n$ operations.

This is a huge improvement over linear search.

n	$\log_e n$
10	2.3
100	4.6
1000	6.9
1,000,000	13.8
1,000,000,000	20.7

Analysis of Recursive Fibonacci Method

The running time for computing `fibo`(n) can be specified as the recurrence relation:

$$T(n) = \begin{cases} C_2 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + C_1 & \text{otherwise} \end{cases}$$

It can be shown that $T(n)$ is $O((\frac{1+\sqrt{5}}{2})^n)$,
i.e. the running time of the method is *exponential*.

To get an intuition for this, note that the number of recursive calls for `fibo`(n) is close to the number of nodes in a full binary tree of height n, which is

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^{n-1} = 2^n - 1$$

This is called a geometric progression. Since some operations must be performed for each call, the running time must be exponential.

The Dutch National Flag Problem [Dijkstra]

Given an array of `char` containing the characters `'R'` (red), `'W'` (white), and `'B'` (blue) in any order, e.g.

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
 R  B  W  W  B  B  R  W  W  R  R  W  R  B  W
```

write a method to rearrange the array elements so that they appear as in the Dutch national flag, i.e. all reds to the left, all whites in the middle, and all blue to the right:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
 R  R  R  R  R  W  W  W  W  W  W  B  B  B  B
```

The method's running time should be at most linear in the size of the array.

Solution to Dutch National Flag Problem

```
public static void dnf(char[] a)
{  int r = 0;
   int w = 0;
   int b = a.length - 1;
   while (w <= b)
   {  if (a[w] == 'W')
         w++;
      else if (a[w] == 'R')
      {  if (r != w)
            swap(a, r, w);
         r++;
         w++;
      }
      else // if a[w] == 'B'
      {  swap(a, w, b);
         b--;
      }
}
public static void swap(char[] a, int i, int j)
{  int temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```


Running Time of `dnf`

The method's running time is $O(n)$ where $n = \texttt{a.length}$, because the loop does n iterations and does a constant number of operations in each iteration. To see this, note that $\texttt{b} - \texttt{w} + 1$ is equal to n initially, and decreases by 1 at each iteration since either `w` is incremented or `b` is decremented.

Proof of correctness of `dnf`

It is easy to make a mistake in such an algorithm, so we should prove that it is correct. To do this, we must identify a *loop invariant*. As discussed in week 4, this is a condition that is preserved by the loop body, i.e., if it is true at the beginning of a loop iteration and the loop's test condition is true, then the invariant will also be true at the end of the loop iteration.

For our example, a suitable invariant is

$$\texttt{r} \leq \texttt{w} \wedge \texttt{a}[0..\texttt{r}-1] = \texttt{'R'} \wedge$$
$$\texttt{a}[\texttt{r}..\texttt{w}-1] = \texttt{'W'} \wedge \texttt{a}[\texttt{b}+1..\texttt{a.lenght}-1] = \texttt{'B'}$$

where $\texttt{a}[\texttt{i}..\texttt{j}] = c$ means that

$$\forall \texttt{k}, \texttt{i} \leq \texttt{k} \leq \texttt{j} \rightarrow \texttt{a}[\texttt{k}] = c.$$

To show that the method is correct, we show that the invariant is true at the beginning of the loop (trivial given the way the variables are initialized) and preserved by loop iterations. It follows that if the loop terminates, then the invariant is true at the end and the loop's test condition is false. Together, these conditions imply that the array is properly sorted.

To show that the invariant is preserved by the loop body, there are 3 cases to consider: (1) `a[w] == 'W'`, (2) `a[w] == 'R'`, and (3) `a[w] == 'B'`.

Let us show it for case (3); the other cases are similar. We are given that the invariant and the loop's test condition `w <= b` hold before the loop body is executed. We need to show that the invariant still holds after the loop body.

Since $\texttt{r} \leq \texttt{w}$ holds beforehand and `r` and `w` are not changed by the body, it must still hold afterwards.

Since $\texttt{a}[0..\texttt{r}-1] = \texttt{'R'}$ holds beforehand and $\texttt{r} \leq \texttt{w} \leq b$, then $\texttt{a}[0..\texttt{r}-1]$ is not changed by the swap, and so $\texttt{a}[0..\texttt{r}-1] = \texttt{'R'}$ must hold after the body.

$\texttt{a}[\texttt{r}..\texttt{w}-1] = \texttt{'W'}$ is also preserved by the same argument.

Finally, since $\texttt{a}[\texttt{b}+1..\texttt{a.lenght}-1] = \texttt{'B'}$ holds beforehand and $\texttt{a}[\texttt{w}] = \texttt{'B'}$, then $\texttt{a}[\texttt{b}..\texttt{a.lenght}-1] = \texttt{'B'}$ holds after the swap, and thus $\texttt{a}[\texttt{b}+1..\texttt{a.lenght}-1] = \texttt{'B'}$ must hold after the decrementation of `b`.

We should also prove that the method terminates, and that can be done by an argument similar the one we gave to show the method runs in $O(n)$ time.