Sorting and searching

Alexandra Silva and Henk Barendregt

{alexandra,henk}@cs.ru.nl
http://www.cs.ru.nl/~{alexandra,henk}

Institute for Computing and Information Sciences
Radboud University Nijmegen

7 February 2012

Recap

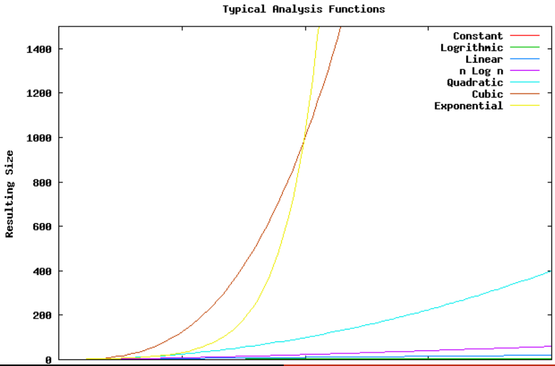
Last week's message

- There are different types of functions
- They have very different growths
- Different algorithms are associated with different functions

Recap

Last week's message

- There are different types of functions
- They have very different growths
- Different algorithms are associated with different functions



Another recap

How to pass this course . . .

- Practice, practice, practice . . .
- You don't learn it it by just staring at the slides - not a spectator sport!
- Exam questions will be in line with exercises

You should come to the werkcollege.

Analysis of algorithms

Study of different features that allow classifying and comparing algorithms

Analysis of algorithms

Study of different features that allow classifying and comparing algorithms

One important feature: the correction of an algorithm. Does the algorithm actually do what we meant it to do? (we will not be dealing with that in this course.)

Analysis of algorithms

Then, it is important to take into account the resources needed to execute an algorithm:

- memory
- bandwidth
- hardware
- ...
- and of particular interest in this course: execution/computation time

The latter is what we will be using as comparison measure between algorithms.

Analysis of algorithms

Then, it is important to take into account the resources needed to execute an algorithm:

- memory
- bandwidth
- hardware
- . . .
- and of particular interest in this course: execution/computation time

The latter is what we will be using as comparison measure between algorithms.
The strategy used in an algorithm to perform a certain task is very important!

In a nutshell. . .

What we will be doing for the next couple of lectures.

- Analysis of execution times (making use of the big-O notation of last lecture):
    - Computational models
    - Analyze the best, worst and average case of an algorithm
    - Analysis of recursive algorithms (recurrence relations)
    - (Analysis of average case; randomized algorithms)
- Analysis of strategies: linear, incremental divide and conquer, (greedy algorithms, dynamic programming)
- Sorting (and searching) algorithms
  Case studies: linear search, binary search, insertion sort, mergesort, quicksort, radix sort, heap sort.

For the sake of clarity: what is an algorithm

An algorithm is a well-defined procedure which takes an input value and returns an output value

Alternative definitions

- a sequence of computational steps transforming an input value into an output value
- a tool to solve a well defined computational problem, which defined the relation between input and output
- . . .

Why are algorithms important?

This was already mentioned before – typical areas of application:

- Internet: routing, searching, etc
- Security and cryptography
- Operations research (have you wondered how TNT programs their deliveries?)
- Maps (TomTom and such)

Why are algorithms important?

This was already mentioned before – typical areas of application:

- Internet: routing, searching, etc
- Security and cryptography
- Operations research (have you wondered how TNT programs their deliveries?)
- Maps (TomTom and such)

Examples of hard problems

- how to find the shortest/fastest path from N (Nijmegen) to A (Amsterdam) in a given map?
- how to visit all the cities in the Netherlands in the shortest way possible?
- given a set of matrices (possibly not square), how to compute their multiplication?

Why are efficient algorithms important?

For a given problem, the ultimate quest is to find the most efficient algorithm.

Why are efficient algorithms important?

For a given problem, the ultimate quest is to find the most efficient algorithm.
For harder problems, no efficient solution is known. These algorithms are NP (we will cover this formally in a few lectures).

Why are efficient algorithms important?

For a given problem, the ultimate quest is to find the most efficient algorithm.
For harder problems, no efficient solution is known. These algorithms are NP (we will cover this formally in a few lectures). A subclass of NP which is interesting is the class of NP-complete problems, because:

- The description of the problem is simple;
- they arise usually in common and important application areas;
- it is not know whether there is an efficient algorithm to solve them;
- if someone finds an efficient solution for one of these problems, then all of them are solvable efficiently!
- when (trying to) solves an NP-complete problem, solutions that come close to the ideal solution are enough, a long as computed in reasonable time.

Why is analysis of algorithms important?

Why should we care to learn how to design and analyze algorithms?

- Even though many problems have been solved efficiently, the rapid change in the world (think of the amount of info) requires for constant improvement in the existing algorithms and design of new, more efficient algorithms;
- If we would have infinite resources (memory, processing speed, . . .) then any solution would be fine; but in reality we have to optimize consumption and minimize the resource consumption (we have to be *zuinig*)!

Why is analysis of algorithms important?

Why should we care to learn how to design and analyze algorithms? (c'd)

- Algorithms for the same problem can have (crazily) different behaviors: good algorithm design can have more impact than the choice of hardware.
- In a nutshell: algorithms are important because the choices made at that level of devising a system can strongly affect its validity and scope of application.

Execution time analysis computational model

To do this type of analysis we need to take to fix our setting:

- Which model should we use?
- What is the cost of the resources in that model

Execution time analysis computational model

To do this type of analysis we need to take to fix our setting:

- Which model should we use?
- What is the cost of the resources in that model

This is what we will use:

- Model: Random Access Machine (RAM).
- A generic computer, single processor, no concurrency (instructions executed sequentially)
- The algorithms are implemented as programs in such a simplified computer

Execution time analysis computational model

We will not specify the model completely; we will be reasonable when it comes to using it. . .

A computer does not have complicated functions as *search* or *sort* but more primitive/atomic ones such as:

- arithmetic operations: sum, multiplication, . . .
- basic data manipulation: store, access, copy, . . .
- basic control structures: if-then-else, loops, . . .

For the purpose of this course, all these primitive operations execute in constant time.

Execution time analysis computational model

Talking about being reasonable:

Do you think exponentiation is executed in constant time?

The RAM computational model does not take into account the memory hierarchy computers have (cache, virtual memory), however the analysis using this model is (almost) always a good representative.

Execution time analysis computational model

Dimension of the input depends on the sort of problem being analyzed:

- sorting a list/vector: number of elements
- multiplying two integers: total number of bits used in the number representation
- shortest path in a graph: the input has two *items* – (V,E) – vertices and edges

Execution time: number of primitive/atomic operations executed

Operations are defined independently of a specific machine

Do you think call of a sub-routine is an atomic operation?

Example I : Linear search in a vector

```
int linsearch(int *v, int a, int b, int k) {
   int i;
   i=a;
   while ((i<=b) && (v[i]!=k))
     i++;
   if (i>b)
      return -1;
   else return i;
 }
```

Example I : Linear search in a vector

```
                                              Cost

int linsearch(int *v, int a, int b, int k) {
   int i;
   i=a;                                       c1
   while ((i<=b) && (v[i]!=k))                c2
     i++;                                     c3
   if (i>b)                                   c4
      return -1;                              c5
   else return i;                             c5
 }
```

Example I : Linear search in a vector

	Cost	repetitions
`int linsearch(int *v, int a, int b, int k) {`		
`   int i;`		
`   i=a;`	c_1	1
`   while ((i<=b) && (v[i]!=k))`	c_2	m+1
`     i++;`	c_3	m
`   if (i>b)`	c_4	1
`      return -1;`	c_5	1
`   else return i;`	c_5	1
`}`		

m is the number of time `i++` executes.

Example I : Linear search in a vector

Code	Cost	repetitions
`int linsearch(int *v, int a, int b, int k) {`		
`  int i;`		
`  i=a;`	c_1	1
`  while ((i<=b) && (v[i]!=k))`	c_2	m+1
`    i++;`	c_3	m
`  if (i>b)`	c_4	1
`     return -1;`	c_5	1
`  else return i;`	c_5	1
`}`		

m is the number of time `i++` executes.

Obs1: These value depends on the while loop condition being true – the position i is in-between a and b, that is: $0 \leq m \leq b - a + 1$.

Total Execution Time

$$T(n) = c_1 + c_2(m+1) + c_3 m + c_4 + c_5$$

Total Execution Time

$$T(n) = c_1 + c_2(m+1) + c_3 m + c_4 + c_5$$

For a given n (the size of the input to search, $n = b - a + 1$), the value $T(n)$ can vary: why?

Total Execution Time

$$T(n) = c_1 + c_2(m+1) + c_3 m + c_4 + c_5$$

For a given n (the size of the input to search, $n = b - a + 1$), the value $T(n)$ can vary: why? Depends on when the element is found.

Total Execution Time

$$T(n) = c_1 + c_2(m+1) + c_3 m + c_4 + c_5$$

For a given n (the size of the input to search, $n = b - a + 1$), the value $T(n)$ can vary: why? Depends on when the element is found.

Best case scenario: The value k is in the first position (`k = v[a]`). Then:

$$T(n) = (c_1 + c_2 + c_4 + c_5)$$

T(n) is constant

Worst case scenario: the value in not found.

$$T(n) = c_1+c_2(n+1)+c_3(n)+c_4+c_5 = (c_2+c_3)n+(c_1+c_2+c_4+c_5)$$

T(n) is linear

Example II : search for duplicates

```
void dup(int *v,int a,int b) {
   int i,j;
   for (i=a;i<b;i++)
     for (j=i+1;j<=b;j++)
       if (v[i]==v[j])
         printf("%d equals to %d\n",i,j);
 }
```

Example II : search for duplicates

```
                                                    Cost

void dup(int *v,int a,int b) {
   int i,j;
   for (i=a;i<b;i++)                                $c_1$
     for (j=i+1;j<=b;j++)                           $c_2$
       if (v[i]==v[j])                              $c_3$
         printf("%d equals to %d\n",i,j);           $c_4$
 }
```

Example II : search for duplicates

```
                                                  Cost      repetitions

void dup(int *v,int a,int b) {
   int i,j;
   for (i=a;i<b;i++)                              c1        N
     for (j=i+1;j<=b;j++)                         c2        S1
       if (v[i]==v[j])                            c3        S2
         printf("%d equals to %d\n",i,j);         c4        S2
 }
```

$N = b - a + 1$ (input size); $S_1 = \sum_{i=a}^{b-1}(n_i + 1)$; $S_2 = \sum_{i=a}^{b-1} n_i$ and $n_i = b - i$ number of times the inner `for` executes, for each i.

Total time of execution

$$T(N) = c_1 N + c_2 S_1 + c_3 S_2 + c_4 S_2$$

Total time of execution

$$T(N) = c_1 N + c_2 S_1 + c_3 S_2 + c_4 S_2$$

In this algorithm the best and worst case coincide: no matter what the size of the input, the `for` cycles are executed the same number of times.

Total time of execution

$$T(N) = c_1 N + c_2 S_1 + c_3 S_2 + c_4 S_2$$

In this algorithm the best and worst case coincide: no matter what the size of the input, the `for` cycles are executed the same number of times.
Let us take $a = 1$ and $b = N$. Then:

$$S_2 = \sum_{i=1}^{N-1} N - i = (N-1)N - \frac{(N-1)N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N$$

Total time of execution

$$T(N) = c_1 N + c_2 S_1 + c_3 S_2 + c_4 S_2$$

In this algorithm the best and worst case coincide: no matter what the size of the input, the `for` cycles are executed the same number of times.

Let us take $a = 1$ and $b = N$. Then:

$$S_2 = \sum_{i=1}^{N-1} N - i = (N-1)N - \tfrac{(N-1)N}{2} = \tfrac{1}{2}N^2 - \tfrac{1}{2}N$$

$$S1 = \sum_{i=1}^{N-1} (N - i + 1) = (N-1)(N+1) - \tfrac{(N-1)N}{2} = \tfrac{1}{2}N^2 - \tfrac{1}{2}N - 1$$

Total time of execution

$$T(N) = c_1 N + c_2 S_1 + c_3 S_2 + c_4 S_2$$

In this algorithm the best and worst case coincide: no matter what the size of the input, the `for` cycles are executed the same number of times.

Let us take $a = 1$ and $b = N$. Then:

$$S_2 = \sum_{i=1}^{N-1} N - i = (N-1)N - \frac{(N-1)N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N$$

$$S1 = \sum_{i=1}^{N-1} (N - i + 1) = (N-1)(N+1) - \frac{(N-1)N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N - 1$$

$T(N)$ is a quadratic function: $T(N) = k_2 N^2 + k_1 N + k_0$.

Some remarks

We simplified the analysis:

- we used abstract costs c_i instead of concrete times

We simplify even further: for a large enough N the term with N^2 is so much larger than the other terms that we can say that the execution time of `dup` is approximated by

$$T(N) = kN^2$$

And in fact k is typically also not relevant. So, using Henk's notation from last time, we can say that the execution time is

$$\Theta(N^2)$$

Some further remarks

Analisys of worst case is useful:

- it's a guarantee: upper limit for any input
- it occurs often (more than one would wish; e.g. search of inexistent information)
- often the average case is very close to the worst case

Some further remarks

Analisys of worst case is useful:

- it's a guarantee: upper limit for any input
- it occurs often (more than one would wish; e.g. search of inexistent information)
- often the average case is very close to the worst case

Analysis of the average case involves (advanced) probabilistic study of the dimension and type of input (e.g. what is the probability a vector is almost sorted?; what is the average size?)

Simplifying further: we will in general assume that, for a given dimension, all inputs occur with the same probability.

Intermezzo

We can classify algorithms in terms of the strategy they use to solve a problem:

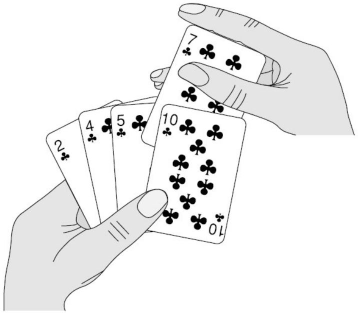
- Incremental – (e.g *insertion sort*, which we will see next).
- Divide and conquer – (e.g *mergesort* and *quicksort*).
- Greedy – (e.g *Minimum Spanning Tree*).
- Dinamyc programming – (e.g *All-Pairs-Shortest- Path*).
- Randomized algorithms – (e.g *randomized quicksort*).

Case study I: Insertion sort

Problem: You're given a list of numbers (integers) and you want to order them. We want to do this on a list given in an array which is to be re-ordered ($\neq$ construct a new list).

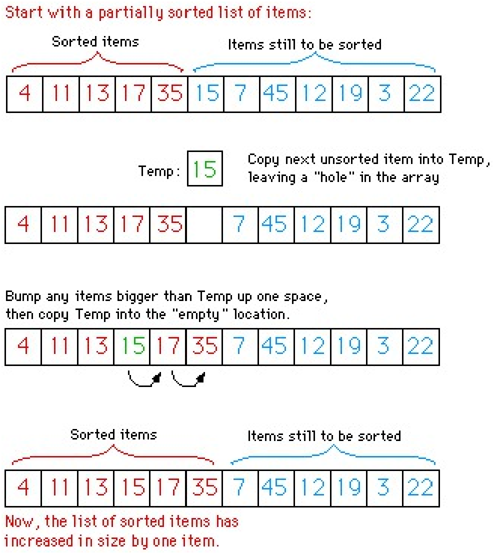
Case study I: Insertion sort

Problem: You're given a list of numbers (integers) and you want to order them. We want to do this on a list given in an array which is to be re-ordered ($\neq$ construct a new list).



- look at the cards
- pick the lower one of a club; put it to the left; search for the next card of that club and continue until you have all the cards inside each club ordered.

Case study I: Insertion sort –example


Start with a partially sorted list of items:
Sorted items
Items still to be sorted
4 11 13 17 35 15 7 45 12 19 3 22
Temp: 15
Copy next unsorted item into Temp, leaving a "hole" in the array
4 11 13 17 35 7 45 12 19 3 22
Bump any items bigger than Temp up one space, then copy Temp into the "empty" location.
4 11 13 15 17 35 7 45 12 19 3 22
Sorted items
Items still to be sorted
4 11 13 15 17 35 7 45 12 19 3 22
Now, the list of sorted items has increased in size by one item.

Case study I: Insertion sort – algorithm

```
void insertion_sort(int A[]) {
   for (j=2 ; j<=N ; j++) {
      key = A[j]; // store the current elm in a tmp record
      i = j-1;
      while (i>0 && A[i] > key) {
     /* search for the position where A[j] belongs */
         A[i+1] = A[i]; /* shift the rest of the greater
                                       elements down */
         i--;
      }
     A[i+1] = key; // place the element
  }
}
```

Case study I: Insertion sort – analysis

Execution time will depend on several things.

- dimension of the list
- how ordered it is already

But let’s again take it a step at a time . . .

Case study I: Insertion sort – analysis

	Cost	repetitions
`void insertion_sort(int A[]) {`		
`  for (j=2 ; j<=N ; j++) {`	c_1	N
`    key = A[j];`	c_2	$N-1$
`    i = j-1;`	c_3	$N-1$
`    while (i>0 && A[i] > key) {`	c_4	S_1
`      A[i+1] = A[i];`	c_5	S_2
`      i--;`	c_6	S_2
`    }`		
`   A[i+1] = key;`	c_7	$N-1$
`  }`		
`}`		

where $S_1 = \sum_{j=2}^{N} n_j$; $S_2 = \sum_{j=2}^{N}(n_j - 1)$, where n_j is the number of times the while is executed, for each value of j

Total execution time

$$T(N) = c_1 N + c_2(N-1) + c_3(N-1) + c_4 S_1 + c_5 S_2 + c_6 S_2 + c_7(N-1)$$

Depending on the input the time can vary.

Best case:

Total execution time

$$T(N) = c_1 N + c_2(N-1) + c_3(N-1) + c_4 S_1 + c_5 S_2 + c_6 S_2 + c_7(N-1)$$

Depending on the input the time can vary.

Best case: The vector is already sorted

$$n_j = 1 \text{ for } j = 2, \ldots, N; \text{ Hence } S_1 = N?1 \text{ and } S_2 = 0;$$

Thus:

$$T(N) = (c_1 + c_2 + c_3 + c_4 + c_7)N - (c_2 + c_3 + c_4 + c_7)$$

Total execution time

$$T(N) = c_1 N + c_2(N-1) + c_3(N-1) + c_4 S_1 + c_5 S_2 + c_6 S_2 + c_7(N-1)$$

Depending on the input the time can vary.

Best case: The vector is already sorted

$$n_j = 1 \text{ for } j = 2, \ldots, N; \text{ Hence } S_1 = N?1 \text{ and } S_2 = 0;$$

Thus:

$$T(N) = (c_1 + c_2 + c_3 + c_4 + c_7)N - (c_2 + c_3 + c_4 + c_7)$$

$T(N)$ linear: in the best case we have an execution time in $\Theta(N)$.

Total execution time

Worst case:

Total execution time

Worst case: The vector is ordered backwards (from the greatest to the smallest element.)

$$n_j = j \text{ for } j = 2, \ldots, N; \text{ Hence}$$

$$S_1 = \sum_{j=2}^{N} j = \frac{N(N+1)}{2} - 1 \text{ and } S_2 = \sum_{j=2}^{N} (j-1) = \frac{N(N-1)}{2};$$

Thus:

$$T(N) = \tfrac{1}{2}(c_4 + c_5 + c_6)N^2 + (c_1 + c_2 + c_3 + .5c_4 - .5c_5 - .5c_6 + c_7)N \\ -(c_2 + c_3 + c_4 + c_7)$$

Total execution time

Worst case: The vector is ordered backwards (from the greatest to the smallest element.)

$$n_j = j \text{ for } j = 2, \ldots, N; \text{ Hence}$$

$$S_1 = \sum_{j=2}^{N} j = \frac{N(N+1)}{2} - 1 \text{ and } S_2 = \sum_{j=2}^{N} (j-1) = \frac{N(N-1)}{2};$$

Thus:

$$T(N) = \tfrac{1}{2}(c_4 + c_5 + c_6)N^2 + (c_1 + c_2 + c_3 + .5c_4 - .5c_5 - .5c_6 + c_7)N \\ -(c_2 + c_3 + c_4 + c_7)$$

$T(N)$ quadratic: worst case has an execution time in $\Theta(N^2)$.

Total execution time

Worst case: The vector is ordered backwards (from the greatest to the smallest element.)

$$n_j = j \text{ for } j = 2, \dots, N; \text{ Hence}$$

$$S_1 = \sum_{j=2}^{N} j = \frac{N(N+1)}{2} - 1 \text{ and } S_2 = \sum_{j=2}^{N} (j-1) = \frac{N(N-1)}{2};$$

Thus:

$$T(N) = \tfrac{1}{2}(c_4 + c_5 + c_6)N^2 + (c_1 + c_2 + c_3 + .5c_4 - .5c_5 - .5c_6 + c_7)N \\ -(c_2 + c_3 + c_4 + c_7)$$

$T(N)$ quadratic: worst case has an execution time in $\Theta(N^2)$.
We can say the time of execution is in $\Omega(N)$ and $O(N^2)$.