Intro to C

Tutorial 11: Search
Last week..

- Connection between pointers and arrays (pointer arithmetic)
- Passing arrays to functions
- Strings
Agenda

• Time complexity
• Example of linear search
• Example of binary search
Measuring time

• Every program that runs on a computer requires two types of resources -

Memory

Time
Computation Time

• You can measure the computation time in the following ways

  Number of instructions executed

  Run time in seconds
Type of processor

• Assume that we run the following code and measure the run time using a clock.

```c
for (i=0; i<10; i++)
{
    sum = sum + i * i;
}
```

• On Moshe’s computer it took 2 seconds.
• On Dana’s computer it took 4 seconds
• What’s the problem with this type of measurement?
Property 1

• Dana’s computer must be slower, so this makes it tough to compare run time.
• Thus, instead of taking time with a clock, we’ll measure the run time as the number of instructions that the program performs.

Time complexity measurement is independent of the processor type.
Size of the input

• Look at the following code fragment:

```c
int sumUp(int a[], int n)
{
    int sum=0, i;
    for (i=0; i<n; i++)
        sum = sum + a[i];
    return sum;
}
```

• How many instructions are executed for an array of size 3?
• And how many for an array of size 1000?
Property 2

• A program’s run time is usually dependent on the size of the input
• The larger the input, the longer the run time.
• We will measure the run time of the program in a manner that is dependent on the input.

Time complexity measurement relates to the input size
In the above examples, the behavior of the code was consistent for all input. What about the next example?

```c
int sum_abs2(int a[], int n)
{
    int sum=0, i;
    for (i=0; i<n; i++)
        sum = sum + (a[i]>0 ? a[i] : -2*a[i]);
    return sum;
}
```

• How many instructions are executed if all of the elements are positive?
• And how many if they are all negative?
Property 3

- For two inputs of the same length, the program might have a different run time!
- Therefore, we define the complexity measurement to be independent of a specific input.
- There are several ways of achieving this. In our course we will put the emphasis on the worst case run time.
  - The input that results in the maximum number of steps (the maximum run time)

Time complexity measurement refers to the worst case scenario
More or less…

• Which code segment is faster? (fewer steps)

```c
int search(int a[], int n, int x)
{
    int i = 0;
    while (i<n) {
        if (a[i] == x) break;
        i++;
    }
    return (i<n)? i : -1;
}
```

```c
int search(int a[], int n, int x)
{
    int i = 0;
    while (i<n) {
        if (a[i] == x) return i;
        i++;
    }
    return -1;
}
```

• What is the difference?
• Is the difference significant?
Property 4

- The number of operations in code depends on the language and the programmer.
- Therefore, we want to measure complexity in a manner that, in addition to measuring complexity of C programs, also lets us measure our algorithm in general.

Time complexity measurement is independent of implementation.
Summary of properties for complexity measurements

1. Independent of processor type
2. Proportional to the input size.
3. Independent of any specific input
4. Independent of implementation.
Asymptotic analysis of complexity

• For the measurement to be independent of implementation, we need to execute an asymptotic approximation. That is to say, we’d like an estimate on the order of magnitude of the complexity.

• **Asymptotic analysis** examines the growth rate of the function.
  
  How does the difficulty of the problem increase when increasing the input?

• Asymptotic analysis helps us determine the behavior of the function.
### Examples of asymptotic behavior

<table>
<thead>
<tr>
<th>Number of steps (as function of the input)</th>
<th>Asymptotic complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$3n + 2$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$7n^3 + 4n + 100$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>$2 \cdot 3^n + n^{50} + 2n$</td>
<td>$O(3^n)$</td>
</tr>
</tbody>
</table>
How will we compare algorithms?

• Sometimes there are many algorithms to solve the same problem.
• Usually, we’ll want to choose the more efficient algorithm.
  – Sometimes additional memory is needed, or there are other special conditions that point us to a less efficient algorithm.
• To choose the most effective algorithm among several options, we look at the asymptotic analysis for each one.
Order of magnitude of functions

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
- $O(n^n)$

Fixed number of steps for each input.

Where will $O(n^4)$ fit in this order? What about $O(\log_3 n)$?
What is the time complexity for each algorithm?

- Computing the min/max of an unsorted array.
- Searching for an element in an unsorted array.
- Computing the min/max of an sorted array.
- Searching for an element in an sorted array.
- Merging sorted arrays
Example: Searching through DNA
Recall: DNA structure

- DNA (molecules that include the hereditary information in living cells) is constructed from four “building blocks” (nucleotides):
  - **Adenine**
  - **Cytosine**
  - **Guanine**
  - **Thymine**

- We’ll use the first letter of the nucleotide (A/C/G/T) to represent it.
Recall: Searching through DNA

- Technion scientists have discovered that people with that have the DNA sequence “ATTAC” are prone to violence
- We will write a program that searches through DNA and will search for this sequence
- **Input:** A sequence of letters
  - End of the sequence will be marked by an X
- **Output:** Whether or not the sequence was found.
Recall: How will we execute the search?

- We are looking for a sequence of 5 characters within a larger sequence (where size is not known in advance)

This is comparable to looking at the strand of DNA with a magnifying glass:
Recall: The original solution

- We previously solved this by maintaining 5 variables of type char.
- Each type a character was received as input, we “moved” the content of the five variables forward.
  - n1 received the old value of n2 and so on.

- The problem with this solution:
  - If we want to search for several different things (for example, ATTAC which measures violent tendencies, as well as CATCAT, which measures growth problems) – things get very complicated…
  - …Because we don’t store all of the data at any point; we only remember the last 5 characters (nucleotides) at a time.
New solution

- New solution: **store input characters in an array**
- Now we can scan the array as many times as we want, each time checking for a different gene

- **Step 1: Input**
- Remember: end of the input is marked with an ‘X’

```c
char gene[1000];
int i = 0, gene_size = 0;

while (i < 1000) {
    scanf("%c", &gene[i]);
    if (gene[i] == 'X')
        break;
    i++;
}
gene_size = i;
```
• Step 2: Function that executes the search for a given gene

```
int search_in_gene(char* gene, int gene_size,
                  char* search, int search_size)
{
    int i, j;
    for (i = 0; i < (gene_size - search_size + 1); i++) {
        int found = 1;
        for (j = 0; j < search_size; j++) {
            if (gene[i+j] != search[j]) {
                found = 0;
                break;
            }
        }
        if (found) return i;
    }
    return -1; /* not found */
}
```
Example using the function

• Assume that the gene and gene_size variables have already been initialized (from code in the first step)
• Now we can search:

```c
char g1[] = {'A', 'T', 'T', 'A', 'C'};
char g2[] = {'C', 'A', 'T', 'C', 'A', 'T'};
int f1;
int f2;

f1 = search_in_gene(gene, gene_size, g1, 5);
if (f1 >= 0)
    printf("ATTAC found at location %d\n", f1);

f2 = search_in_gene(gene, gene_size, g2, 6);
if (f2 >= 0)
    printf("CATCAT found at location %d\n", f2);
```
Time complexity

- What is the time complexity of the function above?
- We’ll use $L$ to denote the length of the gene that we are searching for, and $N$ to describe the length of the DNA sequence.
- For every character in the DNA, we check if it matches the first character in the gene being searched for. There are $N$ such characters.
- For every such location, we check if the $L$ characters of the gene fit.
- Thus, in the worst case there will be $O(N*L)$ operations.
Example: Binary Search
Searching exercise

• Write a function that takes two arrays float a[ ] and float b[], an additional number x, and their size n.
• Each element in a[ ] is different from all others. The same applies for b[].
• It’s known that a[ ] is sorted in decreasing order whereas b[ ] is sorted in increasing order

• The function will return an index i such that:
  \[ 2 \cdot a[i] - 3 \cdot b[i] = x \]
  – If no such index exists, return -1.

• Guidance: The function should be as efficient as possible. That is to say – it is forbidden to execute a linear search in the arrays.
The path to the solution

- **Recall**: Linear search is a search that goes over each element one-by-one.
- A binary search is more efficient, though a **sorted array is required**.
- We’ll check if the expression $2 \cdot a[i] - 3 \cdot b[i]$ results in a sorted sequence of numbers:
  - $a$ is sorted in decreasing order, so as $i$ increases, $a[i]$ gets smaller and reduces the value of the expression.
  - $b$ is sorted in increasing order, so as $i$ increases, $b[i]$ is increased, so this also decreases the resulting expression (because it is subtracted).
    Therefore, the sequence $2 \cdot a[i] - 3 \cdot b[i]$ is sorted in descending order.

<table>
<thead>
<tr>
<th>a: 9</th>
<th>5</th>
<th>4</th>
<th>1</th>
<th>-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b: -4</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2a - 3b: 30</td>
<td>4</td>
<td>-1</td>
<td>-19</td>
<td>-28</td>
</tr>
</tbody>
</table>

- Therefore we’ll use a binary search on $2 \cdot a[i] - 3 \cdot b[i]$. 
int find_index(float a[], float b[], int n, float x)
{
    int low = 0, high = n-1, mid;
    float exp;
    while(low <= high) {
        mid = (low + high)/2;
        exp = 2*a[mid] - 3*b[mid];

        if(exp == x) {
            return mid;
        }
        if(exp > x) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

This is a binary search with a small change: Instead of searching for a value of an array, search for a value of exp that is equal to x

Would it have been possible to build an array c[i]=2*a[i]-3*b[i] and use a binary search without any changes?