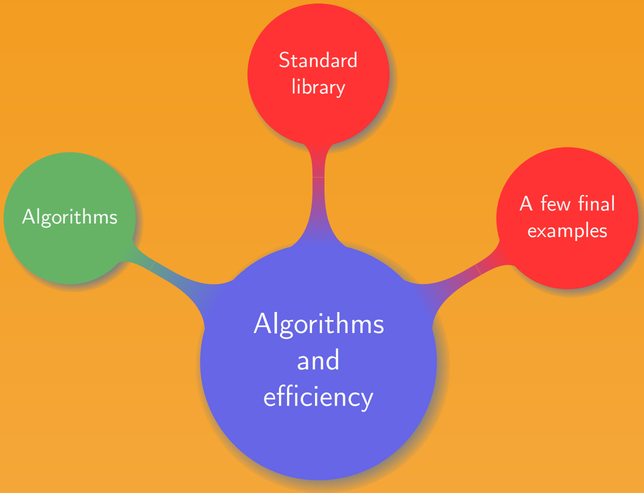
Introduction to Computer and Programming

8. Algorithms and efficiency

Manuel – Summer 2019

Chapter organisation



What is already known

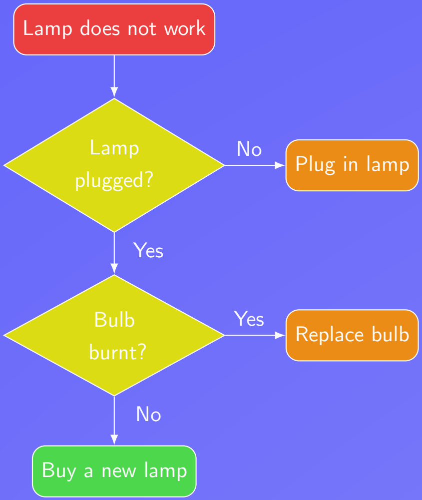
Reminders:

- Algorithms are like recipes for computers
- An algorithm has three main components:
  - Input
  - Output
  - Instructions
- Clear algorithms are often easy to implement
- Algorithms should be adjusted to fit the language
- Algorithms can often be represented as a flowchart

What is already known

Reminders:

- Algorithms are like recipes for computers
- An algorithm has three main components:
    - Input
    - Output
    - Instructions
- Clear algorithms are often easy to implement
- Algorithms should be adjusted to fit the language
- Algorithms can often be represented as a flowchart



Design paradigms

Most common types of algorithms:

- Brute force: often obvious, rarely best
- Divide and conquer: often recursive
- Search and enumeration: model problem using a graph
- Randomized algorithms: feature random choices
  - Monte Carlo algorithms: return the correct answer with high probability
  - Las Vegas algorithms: always correct answer but feature random running times
- Complexity reduction: rewrite a problem into an easier one

Efficiency

When writing a program:

- How efficient does the program need to be?
- What language to choose?
- Is it possible to optimize the code?
- What size are the Input?
- Is it worth implementing a more complex algorithm?

Efficiency

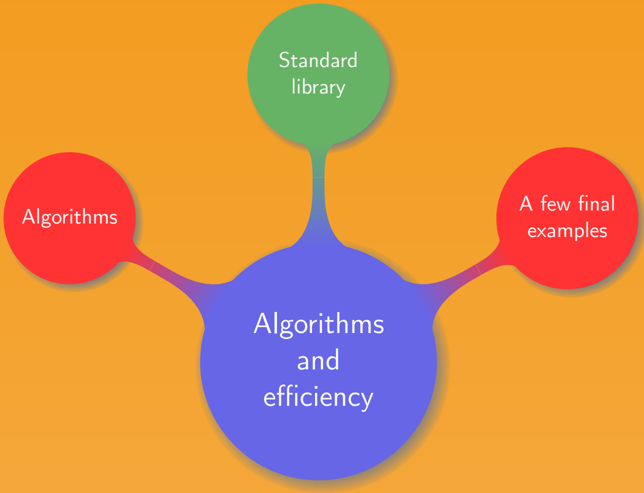
When writing a program:

- How efficient does the program need to be?
- What language to choose?
- Is it possible to optimize the code?
- What size are the Input?
- Is it worth implementing a more complex algorithm?

Computational complexity:

- Evaluates how hard it is to solve a problem
- Independent of the implementation
- Considers the behavior at the infinity
- Both time and space complexity can be considered

Chapter organisation



<stdio.h>

Moving in a file:

- Open a file:
  - `FILE *fopen(const char *path, const char *mode)`
  - `mode` is one of `r`, `r+`, `w`, `w+`, `a`, `a+`
  - `NULL` returned on an error
- Close a file:
  - `int fclose(FILE *fp)`
  - `0` returned on success
- Seek in a file:
  - `int fseek(FILE *stream, long offset, int whence)`
  - `whence` is one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`
- Back to the beginning: `void rewind(FILE *stream)`

<stdio.h>

Reading and writing:

- Write in stream:
```
int fprintf(FILE *stream, const char *format, ...);
```
- Write in string:
```
int sprintf(char *str, const char *format, ...);
```
- Flush a stream: `int fflush(FILE *stream);`
- Read `size-1` characters from a stream:
```
char *fgets(char *s, int size, FILE *stream);
```
- Read next character from `stream` and cast it to an `int`:
```
int getc(FILE *stream);
```

<string.h>

Strings:

- Length of a string: `size_t strlen(const char *s)`
- Copy a string:
  `char *strcpy(char *dest, const char *src)`
- Copy at most *n* bytes of *src*:
  `char *strncpy(char *dest, const char *src, size_t n)`
- Compare two strings:
  - `int strcmp(const char *s1, const char *s2)`
  - Returned value is $< 0,\ 0,\ > 0$, if $s1 < s2,\ s1 = s2,\ s1 > s2$
- Compare the first *n* bytes of two strings:
  `int strncmp(const char *s1, const char *s2, size_t n);`
- Locate a character is a string:
  `char *strchr(const char *s, int c);`

<string.h> and <time.h>

Accessing memory:

- Fill memory with a constant byte:
  ```
  void *memset(void *s, int c, size_t n);
  ```
- Copy memory area, overlap allowed:
  ```
  void *memmove(void *dest, const void *src, size_t n);
  ```
- Copy memory area, overlap not allowed:
  ```
  void *memcpy(void *dest, const void *src, size_t n);
  ```

`<string.h>` and `<time.h>`

Accessing memory:

- Fill memory with a constant byte:

  ```
  void *memset(void *s, int c, size_t n);
  ```

- Copy memory area, overlap allowed:

  ```
  void *memmove(void *dest, const void *src, size_t n);
  ```

- Copy memory area, overlap not allowed:

  ```
  void *memcpy(void *dest, const void *src, size_t n);
  ```

Useful functions for simple benchmarking:

- Getting time: `time_t time(time_t *t);`
- Calculate a time difference:

  ```
  double difftime(time_t time1, time_t time0);
  ```

<ctype.h> and <math.h>

Classifying elements:

- `int isalnum(int c);`
- `int isalpha(int c);`
- `int isspace(int c);`
- `int isdigit(int c);`
- `int islower(int c);`
- `int isupper(int c);`

Converting to uppercase or lowercase:

- `int toupper(int c);`
- `int tolower(int c);`

<ctype.h> and <math.h>

Classifying elements:

- `int isalnum(int c);`
- `int isalpha(int c);`
- `int isspace(int c);`
- `int isdigit(int c);`
- `int islower(int c);`
- `int isupper(int c);`

Converting to uppercase or lowercase:

- `int toupper(int c);`
- `int tolower(int c);`

Common mathematical functions with `double` input and output:

- Trigonometry: `sin(x)`, `cos(x)`, `tan(x)`
- Exponential and logarithm: `exp(x)`, `log(x)`, `log2(x)`, `log10(x)`
- Power and square root: `pow(x,y)`, `sqrt(x)`
- Rounding: `ceil(x)`, `floor(x)`

<stdlib.h>

Mathematics:

- Absolute value: `int abs(int j);`
- Quotient and remainder:
  - `div_t div(int num, int denom);`
  - `div_t`: structure containing two `int`, `quot` and `rem`

Pointers:

- `void *malloc(size_t size);`
- `void *calloc(size_t nobj, size_t size);`
- `void *realloc(void *p, size_t size);`
- `void free(void *ptr);`

<stdlib.h>

Strings:

- String to integer: `int atoi(const char *s);`
- String to long:

```
long int strtol(const char *nptr, char **endptr,
  int base);
```

Misc:

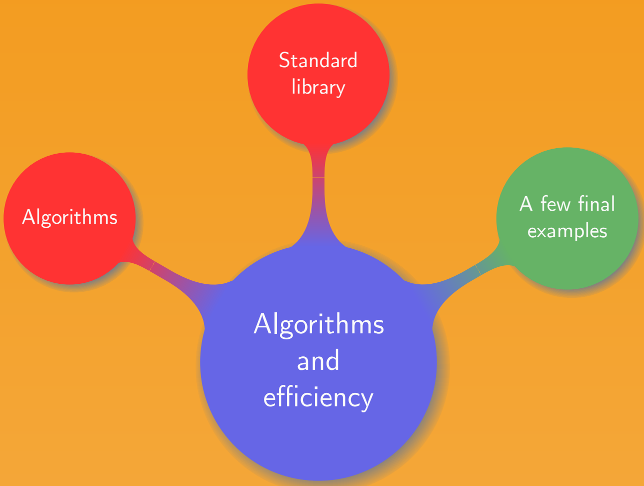
- Execute a system command: `int system(const char *cmd);`
- Sorting:

```
void qsort(void *base, size_t nmemb, size_t size,
  int (*compar)(const void *, const void *));
```

- Searching:

```
void *bsearch(const void *key, const void *base, size_t nmemb,
  size_t size, int (*compar)(const void *, const void *));
```

Chapter organisation



Linear search

linear_search.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 200
#define MAX 1000
int main () {
  int i, n, k=0;
  int data[SIZE];
  srand(time(NULL));
  for(i=0; i<SIZE; i++) data[i]=rand()%MAX;
  n=rand()%MAX;
  for(i=0; i<SIZE; i++) {
    if(data[i]==n) {
      printf("%d found at position %d\n",n,i);
      k++;
    }
  }
  if(k==0) printf("%d not found\n",n);
}
```

Linear search

Adapt the previous code to:

- Read the data from a text file
- Read the value `n` for the standard input
- Exit the program when the first match is found
- Use pointers and dynamic memory allocation instead of arrays

Binary search

binary_search.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 200
int main () {
  int i, n, k=0, low=0, high=SIZE-1, mid;
  int *data=malloc(SIZE*sizeof(int));
  srand(time(NULL));
  for(i=0;i<SIZE;i++) *(data+i)=2*i;
  n=rand()%*(data+i-1);
  while(high >= low) {
    mid=(low + high)/2;
    if(n < *(data+mid)) high = mid - 1;
    else if(n> *(data+mid)) low = mid + 1;
    else {printf("%d found at position %d\n",n,mid);
      free(data);  exit(0);}
  }
  printf("%d not found\n",n);
  free(data);
}
```

Binary search

Using the previous code:

- Write a clear algorithm for binary search
- For a binary search to return a correct result what extra condition should be added on the data?
- Compare the efficiency of a binary search to a linear search; that is on the same data set compare the execution time of the two programs
- Adapt the previous code to use arrays instead of pointers

Selection sort

selection_sort.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 200
#define MAX 1000
int main () {
  int data[SIZE];
  srand(time(NULL));
  for(int i=0; i<SIZE; i++) data[i]=rand()%MAX;
  for(int i=0; i<SIZE; i++) {
    int t, min = i;
    for(int j=i; j<SIZE; j++) if(data[min]>data[j]) min = j;
    t = data[i];
    data[i] = data[min];
    data[min] = t;
  }
  printf("Sorted array: ");
  for(int i=0; i<SIZE; i++) printf("%d ",data[i]);
  printf("\n");
}
```

Selection sort

Understanding the code:

- From the previous code write a clear algorithm describing selection sorting
- How efficient is the selection sort algorithm?
- In the previous program what is the scope of the variables?
- Rewrite the previous code into an independent function
- Generate some unsorted random data and write it in a file; then read the file, sort the data and use a binary search to find a value input by the user

Key points

- Is the most important, the algorithm or the code?
- Cite two types of algorithms
- How is efficiency measured?
- Where to find C functions?

Thank you!