Algorithm Analysis

Portions © S. Hranilovic, S. Dumitrescu and R. Tharmarasa

Department of Electrical and Computer Engineering
McMaster University

January 2020

Textbook Readings

- All of Chapter 5 except Sec. 5.3.3

Efficient Programs?

Question: How do we measure the efficiency of an algorithm ?

- Time required for the program to complete
- Memory space required for the program to operate

Efficient Programs?

Question: How do we measure the efficiency of an algorithm ?

- Time required for the program to complete
- Memory space required for the program to operate

Question: Given two algorithms how do we compare them?

- Implement both algorithms as a program and measure the resources.
  - → this is a very time consuming technique
  - → depends on how well you code each algorithm
  - → how do you choose the test cases to ensure you have a fair measure of performance?
  - → maybe neither algorithm terminates in a reasonable amount of time
- Use the techniques of **algorithm analysis**.

Run Time

- We are often interested in comparing algorithms by the time they require to complete their task.
- There are many factors which influence the speed at which a program runs
  - compiler
  - computer hardware
  - skill of the programmer

Run Time (2)

- We measure the running time of an algorithm by counting the number of basic operations needed for the algorithm to complete, such as:
  - assignments
  - arithmetic operations: add, subtract, multiply, divide, remainder
  - comparisons
- **Justification**: Assume that each basic operation requires the same amount of time units, the time needed for a program to complete is proportional to the number of basic operations.

Run Time (3)

- For most algorithms the **run time** of an algorithm depends on the number of inputs it needs to process.
  - → **Example**: Find the maximum value in a list of n integers.
    - → Input size: n
    - → The number of operations to solve the problem increases as n increases.
- We define the run time required by the algorithm to process n elements as $T(n)$.

Best, Worst and Average Case

- $T_{\text{best}}(n)$ shortest run time over all possible inputs of length n.
- $T_{\text{avg}}(n)$ average run time over all possible inputs of length n.
  - → requires some knowledge about which inputs are most likely
  - → often difficult to find
- $T_{\text{worst}}(n)$ worst run time over all possible inputs of length n.
  - → guarantees a level of performance for algorithm
  - → analysis is easier

$$T_{\text{best}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{worst}}(n)$$

Example - Linear Search

You have an array $A[]$ of length n and you want to know if there is an element of value K. Say each operation cost you c time units.

- $T_{\text{best}}(n) = c$ (the element is the first one)
- $T_{\text{worst}}(n) = cn$ (need to look at every element)

Example - Linear Search (2)

Example: Linear Search - $T_{\text{avg}}(n)$ for successful search

→ Identify all **possible cases**:

→ K found at $A[0]$: 1 comparison

→ K found at $A[1]$: 2 comparisons

...

→ K found at $A[n-1]$: n comparisons

→ Compute the **relative frequency** of each case.

→ If all cases **equally likely**, then the relative frequency of each case is $\frac{1}{n}$.

$$
\begin{aligned}
T_{\text{avg}}(n) &= c \cdot \frac{1}{n} + 2c \cdot \frac{1}{n} + 3c \cdot \frac{1}{n} + \cdots + nc \cdot \frac{1}{n} \\
&= \frac{c}{n} \cdot (1 + 2 + 3 + \cdots + n) \\
&= \frac{c}{n} \cdot \frac{n(n+1)}{2} = \frac{c(n+1)}{2} \\
&\approx \frac{cn}{2}
\end{aligned}
$$

Example - Find largest

Find the largest element in a list of length $n > 0$

```
1  int largest = list[0];          /* current guess are largest element */
2  for (int i=0; i<n; i++)         /* Loop through list */
3      if (list[i] > largest)
4          largest = list[i];      /* store largest number */
5  return largest;
```

- Regardless of list size, need to look at every element in the list!
  - → Line 1 - single operation
  - → Line 2 - one initialization, n increments and $n + 1$ comparisons
  - → Line 3 - executed n times
  - → Line 4 - executed n times (in worst case)
  - → Line 5 - single operation
- **Total number operations**: $T_{\text{worst}}(n) = c(4n + 4)$

Example

```
1  sum = 0;
2  for (i=1; i<=n; i++)
3      for (j=1; j<=n; j++)
4          sum++;
```

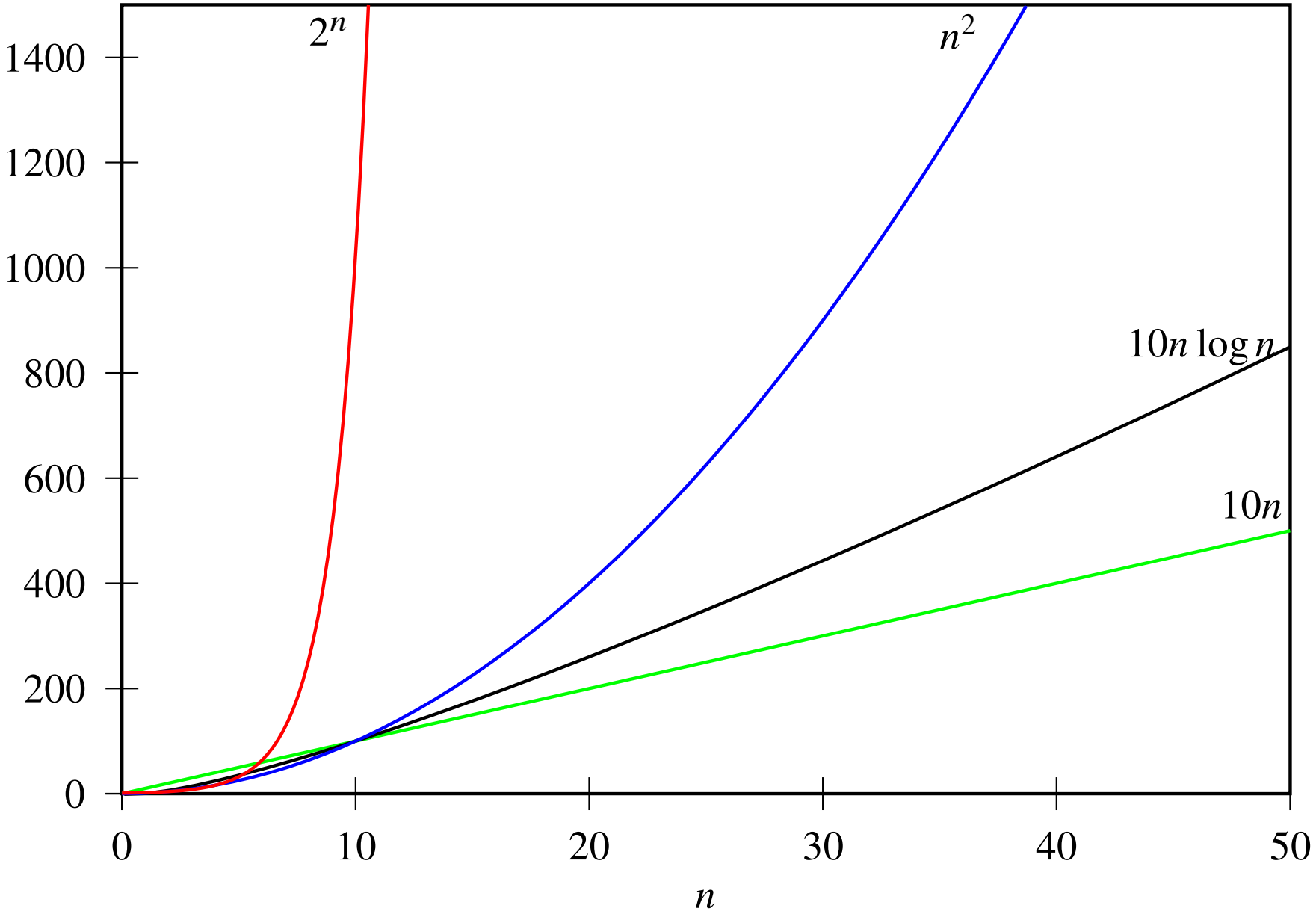
- Line 1 - single operation
- Line 4 - executed n^2 times
- **Total run time**: $T(n) = c_1 n^2 + c_2 n + c_3$, for some positive constants c_1, c_2, c_3.

Asymptotic Analysis

Often we are not interested in the exact run time (since it is too dependent on other factors, e.g., compiler, computer, clock frequency, code implementation, etc.)

- We are interested in how fast $T(n)$ **grows** as n "gets big" - *essential information* on run time.
- If you double the speed of your computer, the relative increase in the number of elements you can process doesn't depend on $T(n)$
- We can ignore the constants c in our asymptotic analysis.
  - → Focus on the **dominant terms** and **shape** of $T(n)$ as n gets large.
- Want to be able to anticipate the requirements of our algorithm as the input size varies.

Growth Rates



Relative Growth Rates

Basic Idea: For a given run time $T(n)$ want to find a simple function $g(n)$ which has the same behaviour as $T(n)$ when n gets large. Consider **relative growth rate**

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Relative Growth Rates

Basic Idea: For a given run time $T(n)$ want to find a simple function $g(n)$ which has the same behaviour as $T(n)$ when n gets large. Consider **relative growth rate**

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- If limit $= 0 \Longrightarrow$ growth rate of $f(n)$ is strictly smaller than $g(n)$
  - e.g., $f(n) = 1000n$ and $g(n) = n^2$

Relative Growth Rates

Basic Idea: For a given run time $T(n)$ want to find a simple function $g(n)$ which has the same behaviour as $T(n)$ when n gets large. Consider **relative growth rate**

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- If limit $= 0 \Rightarrow$ growth rate of $f(n)$ is strictly smaller than $g(n)$
  - e.g., $f(n) = 1000n$ and $g(n) = n^2$
- If limit $\to \infty \Rightarrow$ growth rate of $f(n)$ is strictly larger than $g(n)$
  - e.g., $f(n) = n^3$ and $g(n) = \log n$

Relative Growth Rates

Basic Idea: For a given run time $T(n)$ want to find a simple function $g(n)$ which has the same behaviour as $T(n)$ when n gets large. Consider **relative growth rate**

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- If limit $= 0 \Rightarrow$ growth rate of $f(n)$ is strictly smaller than $g(n)$
  - e.g., $f(n) = 1000n$ and $g(n) = n^2$
- If limit $\to \infty \Rightarrow$ growth rate of $f(n)$ is strictly larger than $g(n)$
  - e.g., $f(n) = n^3$ and $g(n) = \log n$
- If limit $= c$, $c > 0 \Rightarrow$ growth rates of $f(n)$ and $g(n)$ are the same
  - e.g., $f(n) = n^4 + 3n^2$ and $g(n) = 10n^4 + n + 3$

Asymptotic Notation

- Group functions into **complexity classes** according to their growth rate.
- Let $\Theta(g(n))$ denote the "class" of all functions with the same growth rate as $g(n)$.
- **Examples** of functions in the complexity class $\Theta(n)$:
  - $T_1(n) = 30n + 5$;
  - $T_2(n) = \frac{3}{2}n + 2\log_2 n$;
  - $T_3(n) = \sqrt{n} + \frac{1}{2}n$.
- If $f(n)$ is in $\Theta(g(n))$ then
  - Simply write: $f(n) = \Theta(g(n))$
  - Read: "f **of** n **is Big-theta of** g **of** n";
  - or read "f **of** n **is on the order of** g **of** n".

Asymptotic Analysis – Big-Theta

Basic Idea: Find the simplest $g(n)$ so that $T(n) = \Theta(g(n))$

→ Focus on **dominant term** of $T(n)$, the one which grows the fastest.

→ Discard other terms and positive constant multipliers.

Asymptotic Analysis – Big-Theta

Basic Idea: Find the simplest $g(n)$ so that $T(n) = \Theta(g(n))$

→ Focus on **dominant term** of $T(n)$, the one which grows the fastest.

→ Discard other terms and positive constant multipliers.

Example: $T(n) = 2n^2 + n + \sqrt{n} = \Theta(?)$

→ **Identify** Dominant term: n^2

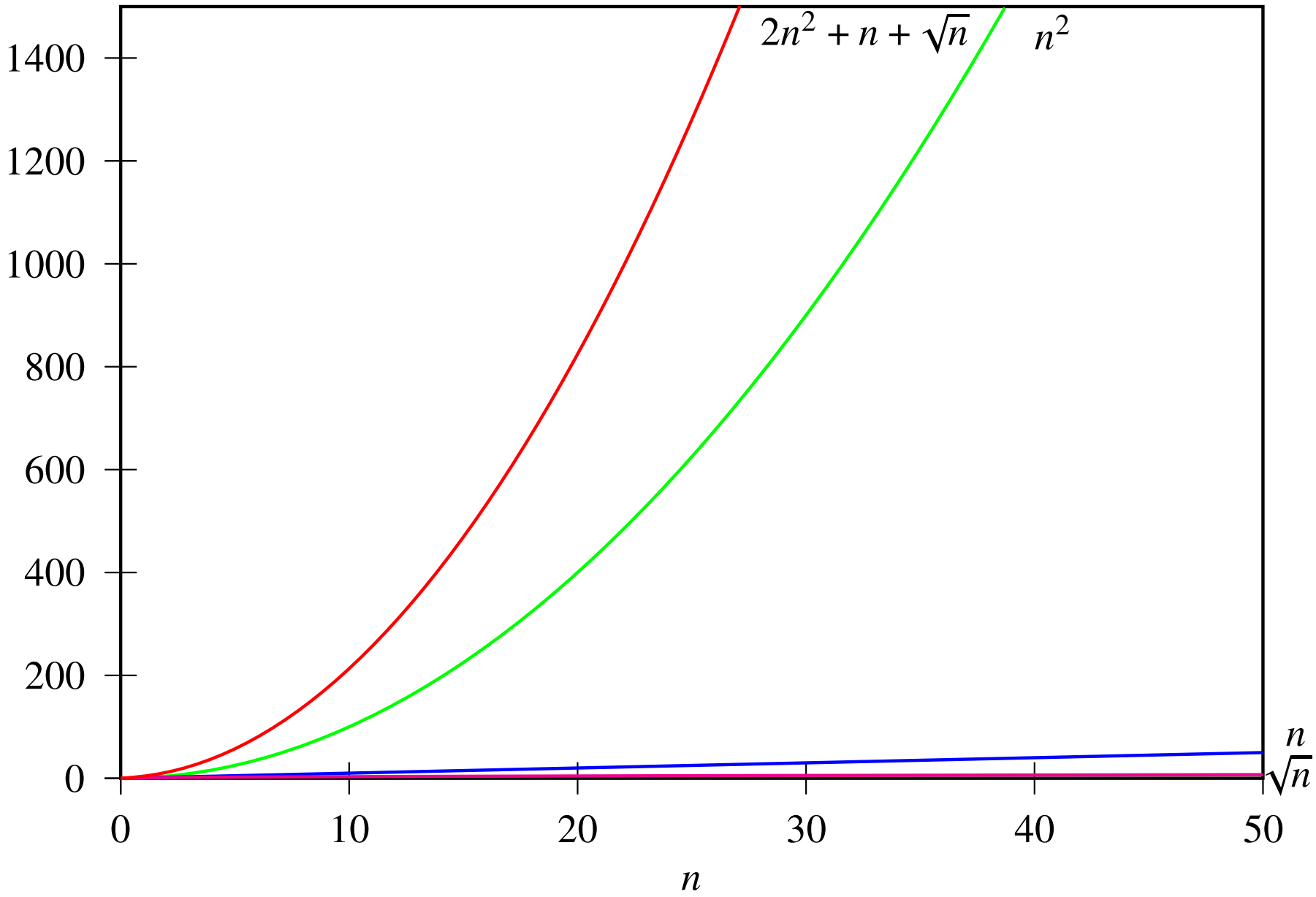
→ $\lim_{n\to\infty} \frac{2n^2}{n} = \infty$,

→ $\lim_{n\to\infty} \frac{2n^2}{\sqrt{n}} = \infty$.

→ **Discard** positive multiplier 2.

→ **Conclude:** $T(n) = \Theta(n^2)$.

Dominant Term



Big-Theta: Example 2

- $T(n) = 12n^2 + n\log n + \frac{1}{6}n^3 + \log\log n = \Theta(?)$

Big-Theta: Example 2

- $T(n) = 12n^2 + n\log n + \frac{1}{6}n^3 + \log\log n = \Theta(?)$
  - → **Identify** dominant term: $\frac{1}{6}n^3$.
  - → **Discard** positive multiplier $\frac{1}{6}$.
  - → **Conclude:** $T(n) = \Theta(n^3)$.

How to use “Big-Theta" information?

Consider we have two algorithms and we determine the following:

- Algorithm A: $T_A(n) = \Theta(\sqrt{n})$
- Algorithm B: $T_B(n) = \Theta(n)$

Which algorithm is **more efficient**, i.e. **faster**, as $n \to \infty$?

How to use "Big-Theta" information? (2)

- $T_A(n) = \Theta(\sqrt{n}) \Rightarrow T_A(n)$ has the **same growth rate** as $\sqrt{n}$ **(1)**
- $T_B(n) = \Theta(n) \Rightarrow T_B(n)$ has the **same growth rate** as n **(2)**
  - Compare growth rates of $\sqrt{n}$ and n.
    - $\lim_{n\to\infty} \frac{\sqrt{n}}{n} = \lim_{n\to\infty} \frac{1}{\sqrt{n}} = \mathbf{0}$.
    - $\sqrt{n}$ has a **strictly smaller** growth rate than n **(3)**
  - From **(1),(2),(3)** it follows that $T_A(n)$ has a **strictly smaller** growth rate than $T_B(n)$, i.e., $\lim_{n\to\infty} \frac{T_A(n)}{T_B(n)} = 0$.
  - Therefore, for n **sufficiently large** we have $T_A(n) < T_B(n)$
- **Conclusion**: Algorithm A is **more time efficient**, than B, as $n \to \infty$.

Complexity Classes

- $\Theta(1)$ **constant complexity** independent of n
- $\Theta(\log n)$ **logarithmic complexity** a constant amount of time is required to cut the problem by a factor (usually 1/2)
- $\Theta(n)$ **linear complexity** constant amount of time is required to cut the problem by a constant amount.
- $\Theta(n \log n)$
- $\Theta(n^2)$ **quadratic complexity**
- $\Theta(n^k)$ **polynomial complexity**
- $\Theta(c^n)$ (for $c > 1$) **exponential complexity** really bad !!

Asymptotic Notation - Upper bound

Upperbound: Sometimes a weaker upper bound on growth rate can be more easily found.

Asymptotic Notation - Upper bound

Upperbound: Sometimes a weaker upper bound on growth rate can be more easily found.

Definition (Big-Oh)

A non-negative function $T(n)$ is said to be **of order** $g(n)$ if there are positive constants c and n_0 such that

$$T(n) \leq cg(n)$$

whenever $n > n_0$. We then write that $T(n) = O(g(n))$

If $T(n) = O(g(n))$ then,

- the growth of $T(n)$ at larger n is **upper bounded** by $g(n)$
- $T(n)$ grows no faster than $g(n)$ for large n.
- In general, we want the tightest upper bound.

Asymptotic Notation - Upper bound (2)

Let $f(n), g(n)$ be positive functions such that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ **exists**

→ If $f(n)$ **is in the complexity class** $O(g(n))$ then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$$

→ $g(n)$ either grows at the **same rate** or **faster than** $f(n)$

→ **Examples**: $n/2 = O(n^2), n/2 = O(n)$.

→ $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$ implies that there is some constant c such that

$$\begin{aligned} \frac{f(n)}{g(n)} &\leq c \quad \text{for } n \text{ sufficiently large} \\ \Rightarrow f(n) &\leq cg(n) \quad \text{for } n \text{ sufficiently large} \end{aligned}$$

Asymptotic Notation - Lower bound

McMaster University

Definition (Big-Omega)

A non-negative function $T(n) = \Omega(g(n))$ if there are positive constants c and n_0 such that

$$T(n) \geq cg(n)$$

whenever $n > n_0$.

If $T(n) = \Omega(g(n))$ then,

- the growth of $T(n)$ at larger n is **lower bounded** by $g(n)$
- $T(n)$ grows at least as fast as $g(n)$ for large n.
- In general, we want the greatest lower bound
- **Examples**: $n/2 \neq \Omega(n^2)$, $n/2 = \Omega(n)$, $n/2 = \Omega(\sqrt{n})$.

Asymptotic Notation - General Big-Theta Definition

McMaster University

Definition (Big-Theta)

A non-negative function $T(n) = \Theta(g(n))$ if $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$.
The function $g(n)$ is a **tight bound**

McMaster University

Definition (Big-Theta)

A non-negative function $T(n) = \Theta(g(n))$ if $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$. The function $g(n)$ is a **tight bound**

Definition (Big-Theta)

A non-negative function $T(n) = \Theta(g(n))$if there are positive constants c_1, c_2 and n_0 such that

$$c_1 g(n) \leq T(n) \leq c_2 g(n)$$

whenever $n > n_0$.

Asymptotic Notation - Weak Upper bound

McMaster University

Definition (Little-Oh)

A non-negative function $T(n) = o(g(n))$ if $T(n) = O(g(n))$ and $T(n) \neq \Theta(g(n))$.
The function $g(n)$ is a **weak bound**

Example:

Consider the function,

$$f(n) = 2n^2 + n + \sqrt{n}$$

- $f(n) = O(n^2)$,

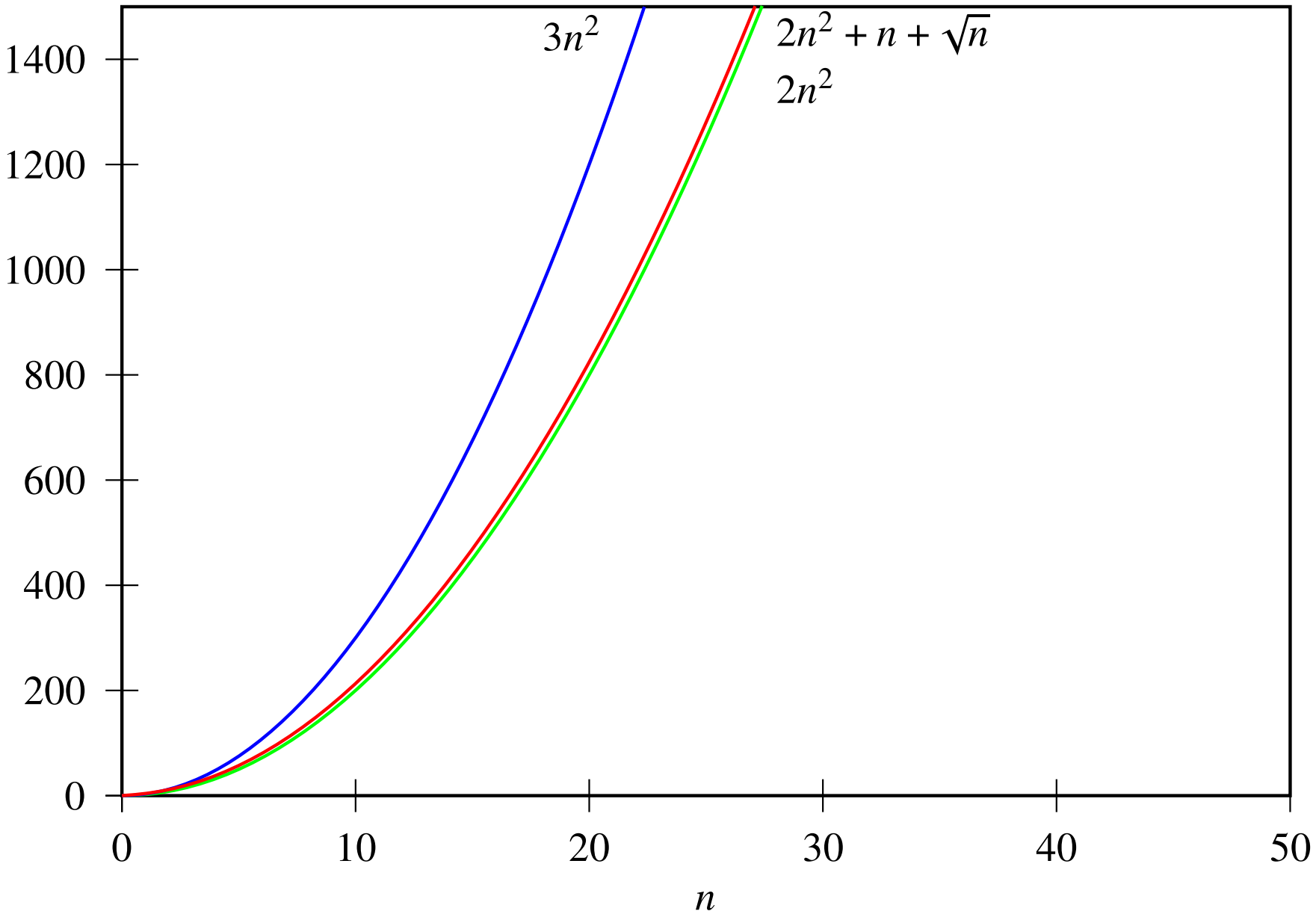
$$\begin{aligned} f(n) &= 2n^2 + n + \sqrt{n} \\ &< 2n^2 + n^2 + n^2, \qquad n > 1 \\ &= 4n^2 \\ &= O(n^2) \end{aligned}$$

- $f(n) = \Omega(n^2)$,

$$\begin{aligned} f(n) &= 2n^2 + n + \sqrt{n} \\ &> 2n^2, \qquad n > 0 \\ &= \Omega(n^2) \end{aligned}$$

- Therefore, $f(n) = \Theta(n^2)$
- $f(n) = o(n^3)$ since $f(n) = O(n^3)$ and $f(n) \neq \Omega(n^3)$.

Asymptotic Notation



Relative Growth Rates

Using asymptotic notation, can also define **relative growth rate** as

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

$$\begin{array}{lll} \text{If} & \text{limit} = 0, & f(n) = o(g(n)) \\ & \text{limit} = c,\ c \neq 0, & f(n) = \Theta(g(n)) \\ & \text{limit} \to \infty, & g(n) = o(f(n)) \end{array}$$

Run Time Calculations

Rules:

- If $T_1(n) = \Theta(f(n))$ and $T_2(n) = \Theta(g(n))$,
  - $T_1(n) + T_2(n) = \Theta(\max\{f(n), g(n)\})$
  - $T_1(n) \times T_2(n) = \Theta(f(n)g(n))$
- If $T(n)$ is a polynomial of degree k, then $T(n) = \Theta(n^k)$
- **Loops**: Run time is the run time of statements in loop times number of iterations (when run time of each iteration is the same)
- **Nested loops**: Run time is product of sizes of all loops and time of innermost statement.
  - **Tip**: Analyze inside out.
- **Consecutive Statements**: Take dominant run time
- **`if` statements**: run time is at most maximum run time in either branch.

Example 1

Assignment operation

```
1  a = b;
```

Run Time Analysis:

Example 1

Assignment operation

```
1  a = b;
```

Run Time Analysis:

$$O(1), \Omega(1) \text{ and } \Theta(1)$$

- Since assignment operation takes a constant amount of time.

Example 2

Simple `for` Loop

```
1 sum = 0;
2 for (i=1; i<=2*n+1; i++)
3     sum += n;
```

Run Time Analysis:

Example 2

McMaster University

Simple `for` Loop

```
1  sum = 0;
2  for (i=1; i<=2*n+1; i++)
3      sum += n;
```

Run Time Analysis:

- Line 1: $T_1(n) = \Theta(1)$.
- `for` loop on lines 2,3: $T_{2,3}(n)$
  - → Number of iterations: $2n+1 = \Theta(n)$
  - → Each iteration (line 3, incrementing i, comparison): constant time= $\Theta(1)$.
  - → Use **Product Rule**: $T_{2,3}(n)$ = number iterations $\times$ run time each iteration $= \Theta(n \times 1) = \Theta(n)$.
- Use **Sum Rule**: $T(n) = T_1(n) + T_{2,3}(n) = \Theta(1+n) = \Theta(n)$.

Example 3

Multiple and Nested for Loops

```
1  sum = 0;
2  for (j=1; j<=n; j++)          /* First Loop */
3      for (i=1; i<=3*n; i++)
4          sum++;
5  for (k=n-1; k>=0; k--)          /* Second Loop */
6      A[k] = k;
```

Run Time Analysis:

Example 3

Multiple and Nested for Loops

```
1  sum = 0;
2  for (j=1; j<=n; j++)              /* First Loop */
3      for (i=1; i<=3*n; i++)
4          sum++;
5  for (k=n-1; k>=0; k--)            /* Second Loop */
6      A[k] = k;
```

Run Time Analysis:

- Line 1: $T_1(n) = \Theta(1)$ (constant time).
- Loop 2: $T_{5,6}(n) = \Theta(n)$ (n iterations, each takes constant time).

Example 3 (2)

Multiple and Nested for Loops

```
1  sum = 0;
2  for (j=1; j<=n; j++)              /* First Loop */
3      for (i=1; i<=3*n; i++)
4          sum++;
5  for (k=n-1; k>=0; k--)            /* Second Loop */
6      A[k] = k;
```

Run Time Analysis:

- Loop 1: $T_{2,3,4}(n)$
  - → Num. iterations outer loop $\times$ Num. iterations inner loop $\times$ Operations inside inner loop
- $T_{2,3,4}(n) = n \times 3n \times 1 = \Theta(n^2)$

$$T(n) = T_1(n) + T_{2,3,4}(n) + T_{5,6}(n) = \Theta(1 + n^2 + n) = \Theta(n^2).$$

Example 4

```
1  sum1 = 0;
2  for (i=1; i<=n; i++)                    /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (i=1; i<=n; i++)                    /* Second Loop */
7      for (j=1; j<=i; j++)
8          sum2++;
```

Run Time Analysis:

Example 4

```
1  sum1 = 0;
2  for (i=1; i<=n; i++)                    /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (i=1; i<=n; i++)                    /* Second Loop */
7      for (j=1; j<=i; j++)
8          sum2++;
```

Run Time Analysis:

- Line 1: $T_1(n) = \Theta(1)$
- **Loop 1**: $T_{2,3,4} = \Theta(n^2)$ (like in previous example).
- Line 5: $T_5(n) = \Theta(1)$

Example 4 (2)

```
1  sum1 = 0;
2  for (i=1; i<=n; i++)                    /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (i=1; i<=n; i++)                    /* Second Loop */
7      for (j=1; j<=i; j++)
8          sum2++;
```

Run Time Analysis:

- Loop 2: $T_{6,7,8}(n)$ (analyze **inside out**)
  - → Line 8: takes constant time c
  - → Line 7: loop runs i times
    - → inner `for` statement running time: ci.
  - → Line 6: outer loop: i takes the values: $1, 2, \cdots, n$

$$
\begin{aligned}
T_{6,7,8}(n) &= \sum_{i=1}^{n} ci = c \cdot 1 + c \cdot 2 + c \cdot 3 + \cdots + c \cdot n \\
&= c(1 + 2 + \cdots + n) = c\frac{n(n+1)}{2} = \frac{c}{2}n^2 + \frac{c}{2}n \\
&= \mathbf{\Theta(n^2)}
\end{aligned}
$$

Example 4 (3)

```
1  sum1 = 0;
2  for (i=1; i<=n; i++)                    /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (i=1; i<=n; i++)                    /* Second Loop */
7      for (j=1; j<=i; j++)
8          sum2++;
```

Run Time Analysis:

- Overall complexity:

$$T(n) = T_1(n) + T_{2,3,4}(n) + T_5(n) + T_{6,7,8}(n) = \Theta(1 + n^2 + 1 + n^2) = \Theta(n^2).$$

- Notice the execution time is $\mathbf{\Theta(n^2)}$, even though loop 2 takes about half the time of loop 1.

Example 5

☞ Assume $n = 2^i$

```
1  sum1 = 0;
2  for (k=1; k<=n; k*=2)              /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (k=1; k<=n; k*=2)              /* Second Loop */
7      for (j=1; j<=k; j++)
8          sum2++;
```

Run Time Analysis:

Example 5

☞ Assume $n = 2^i$

```
1  sum1 = 0;
2  for (k=1; k<=n; k*=2)              /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (k=1; k<=n; k*=2)              /* Second Loop */
7      for (j=1; j<=k; j++)
8          sum2++;
```

Run Time Analysis:

- **Loop 1**: $T_{2,3,4}(n)$
- Line 4: constant time $\Theta(1)$
- Line 3: executed n times

Example 5 (2)

☞ Assume $n = 2^i$

```
1  sum1 = 0;
2  for (k=1; k<=n; k*=2)              /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (k=1; k<=n; k*=2)              /* Second Loop */
7      for (j=1; j<=k; j++)
8          sum2++;
```

Run Time Analysis:

- If $n = 2^i \Rightarrow i = \log_2 n$
- Outer loop variable k takes the values: $1, 2, 2^2, 2^3, \cdots, 2^i$
  - → Total of $i + 1 = \log_2 n + 1$ iterations.

Example 5 (3)

- To find $T_{2,3,4}(n)$ **sum up** running times of inner loop for all k's.

$$\sum_{s=0}^{\log_2 n} cn = cn(1 + \log_2 n)$$

- $T_{2,3,4}(n) = \Theta(n \log n)$.

Example 5 (4)

☞ Assume $n = 2^i$

```
1  sum1 = 0;
2  for (k=1; k<=n; k*=2)              /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (k=1; k<=n; k*=2)              /* Second Loop */
7      for (j=1; j<=k; j++)
8          sum2++;
```

Run Time Analysis:

➠ **Loop 2**: $T_{6,7,8}(n)$

→ Line 8: $\Theta(1)$

→ Line 7: Runs k times for $k = 1, 2, 2^2, 2^3, \cdots, 2^i$

Example 5 (5)

→ Line 6: **Sum up** running times for all k's (and noting that $i = \log_2 n$):

$$\begin{aligned}
T_{6,7,8}(n) &= (c + c2^1 + c2^2 + \cdots c2^i) \\
&= c(1 + 2^1 + 2^2 + \cdots + 2^i) \quad \text{(geometric series)} \\
&= c(2^{i+1} - 1) \\
&= c(2^{\log_2 n + 1} - 1) \\
&= 2cn - c \\
&= \Theta(n).
\end{aligned}$$

Example 5 (6)

☞ Assume $n = 2^i$

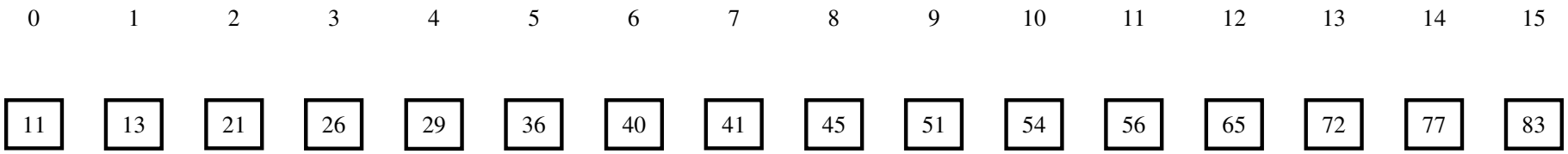
```
1  sum1 = 0;
2  for (k=1; k<=n; k*=2)              /* First Loop */
3      for (j=1; j<=n; j++)
4          sum1++;
5  sum2 = 0;
6  for (k=1; k<=n; k*=2)              /* Second Loop */
7      for (j=1; j<=k; j++)
8          sum2++;
```

Run Time Analysis:

➠ **Overall complexity**:

$$\begin{aligned} T(n) &= T_1(n) + T_{2,3,4}(n) + T_5(n) + T_{6,7,8}(n) = \Theta(1 + n\log n + 1 + n) \\ &= \Theta(n \log n). \end{aligned}$$

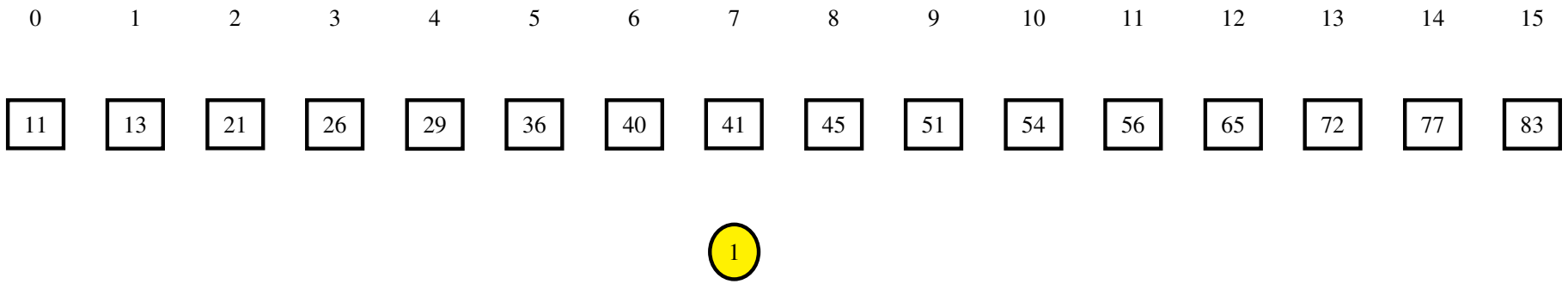
Binary Search



Search for K in a pre-sorted list of length n.

```
1  int l = left-1; int r = right+1; /* l and r are beyond array bounds */
2  while (l+1 != r) {                /* loop until l and r meet */
3      int i = (l+r)/2;              /* find middle of sub-list */
4      if (K < array[i]) r = i;      /* Value in left half */
5      if (K == array[i]) return i;  /* Found it! */
6      if (K > array[i]) l = i;      /* Value in right half */
7  }
8  return UNSUCCESSFUL;              /* Not in array */
```

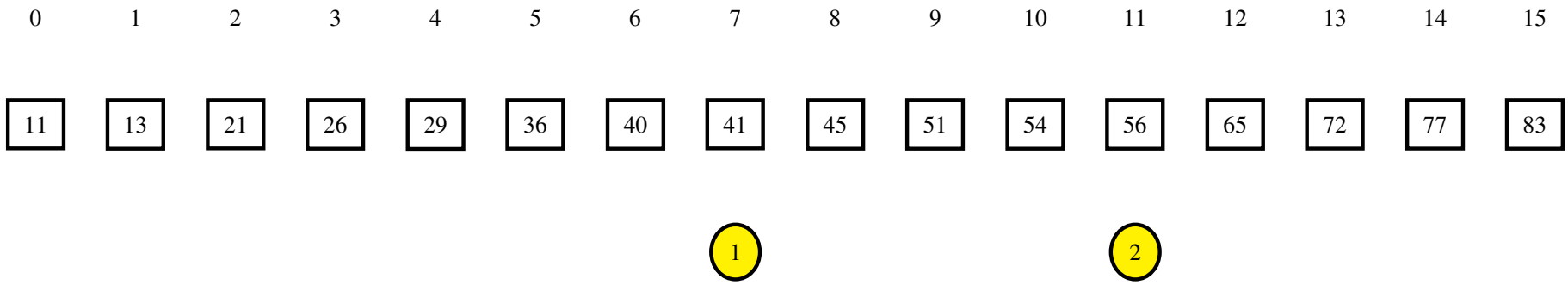
Binary Search



Search for K in a pre-sorted list of length n. **Example:** Search for K=45.

```
1  int l = left-1; int r = right+1; /* l and r are beyond array bounds */
2  while (l+1 != r) {                /* loop until l and r meet */
3      int i = (l+r)/2;              /* find middle of sub-list */
4      if (K < array[i]) r = i;      /* Value in left half */
5      if (K == array[i]) return i;  /* Found it! */
6      if (K > array[i]) l = i;      /* Value in right half */
7  }
8  return UNSUCCESSFUL;              /* Not in array */
```

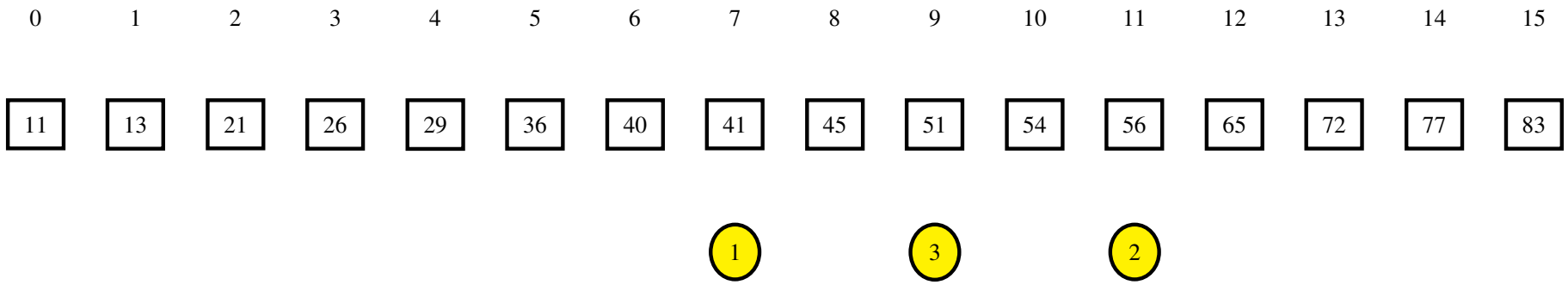
Binary Search



Search for K in a pre-sorted list of length n. **Example:** Search for K=45.

```
1  int l = left-1; int r = right+1; /* l and r are beyond array bounds */
2  while (l+1 != r) {                /* loop until l and r meet */
3      int i = (l+r)/2;              /* find middle of sub-list */
4      if (K < array[i]) r = i;      /* Value in left half */
5      if (K == array[i]) return i;  /* Found it! */
6      if (K > array[i]) l = i;      /* Value in right half */
7  }
8  return UNSUCCESSFUL;              /* Not in array */
```

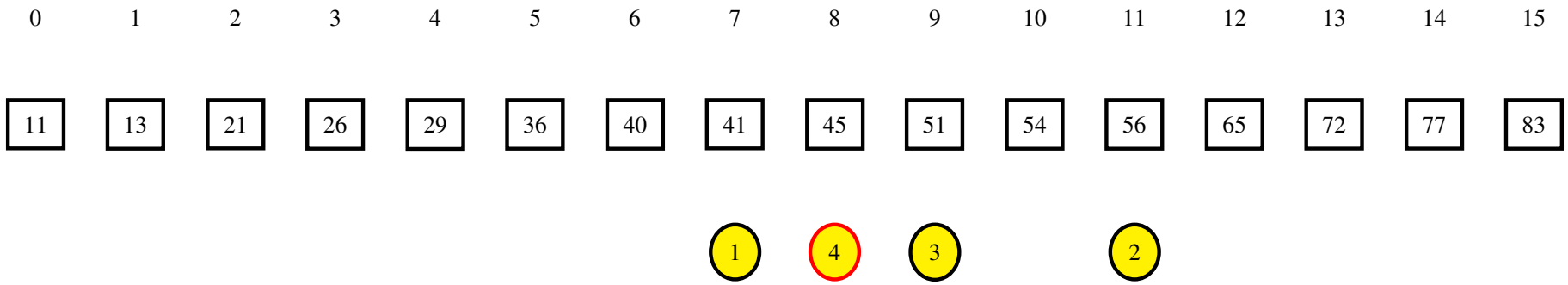
Binary Search



Search for K in a pre-sorted list of length n. **Example:** Search for K=45.

```
1  int l = left-1; int r = right+1; /* l and r are beyond array bounds */
2  while (l+1 != r) {                /* loop until l and r meet */
3      int i = (l+r)/2;              /* find middle of sub-list */
4      if (K < array[i]) r = i;      /* Value in left half */
5      if (K == array[i]) return i;  /* Found it! */
6      if (K > array[i]) l = i;      /* Value in right half */
7  }
8  return UNSUCCESSFUL;              /* Not in array */
```

Binary Search



Search for K in a pre-sorted list of length n. **Example:** Search for K=45.

```
1  int l = left-1; int r = right+1; /* l and r are beyond array bounds */
2  while (l+1 != r) {                /* loop until l and r meet */
3      int i = (l+r)/2;              /* find middle of sub-list */
4      if (K < array[i]) r = i;      /* Value in left half */
5      if (K == array[i]) return i;  /* Found it! */
6      if (K > array[i]) l = i;      /* Value in right half */
7  }
8  return UNSUCCESSFUL;              /* Not in array */
```

Binary Search (2)

Question: What is the worst case run-time for this algorithm ?
Answer: At each iteration of the `while` loop, **half** of the remaining elements in the list are discarded.
For simplicities sake, assume that the number of elements in the list is $n = 2^i$. Then,

$$T(n) = T(n/2) + c, n > 1 \qquad \text{and} \qquad T(1) = c$$

We can only expand this $\log n$ times before we hit the base case.
So, the run-time of this algorithm is,

$$\Theta(\log n)$$

Binary Search (3)

- Note that $\Theta(\log n)$ is much better than the $\Theta(n)$ performance of linear search on the same list.
- Binary search depends on the fact that the array is sorted.
  - Linear search has will be $\Theta(n)$ regardless of list, insertion $\Theta(1)$.
  - Inserting into a sorted list $\Theta(n)$ in the worst case: $\Theta(\log n)$ to find the spot and $\Theta(n)$ to "shuffle" elements to make room for new entry.
- **Tradeoff** between ease of inserting new elements and searching!
  - Which search algorithm you choose will depend on problem definition

Space Complexity

We can analyze the space, i.e., memory, requirements of a program using the asymptotic notation.

Example: What is the memory requirement of an array of n integers?

Say, each integer is represented by c bytes. Therefore, array would require cn bytes $\Rightarrow \Theta(n)$.

Space Complexity

We can analyze the space, i.e., memory, requirements of a program using the asymptotic notation.

Example: What is the memory requirement of an array of n integers?

Say, each integer is represented by c bytes. Therefore, array would require cn bytes $\Rightarrow \Theta(n)$.

Space/Time Tradeoff: It is often possible to get a reduction in time by using more memory resources.

Examples:

- using a look-up table with pre-computed values for complex functions like $\sin(\cdot)$ and $\cos(\cdot)$.
- **Binsort** Say have a permutation of the integers from 0 to $n-1$ in an array `A`. Allocate another array `B` of same size and assign each value to a position equal to its value

```
1  for(i=0 ; i<n ; i++)
2      B[A[i]] = A[i];
```

This algorithm is fast, $\Theta(n)$ in time, however, it requires $\Theta(n)$ extra memory!

Limitations of Asymptotic Algorithm Analysis

- Asymptotic algorithm analysis is relevant when the input size is **"sufficiently"** large.
- For small inputs it is best to use the **simplest** algorithm.
- Sometimes an algorithm is efficient asymptotically, but the "hidden" constant is **too large** to be practical.
- **Example**
  - $T_A(n) = \Theta(n \log n), T_B(n) = \Theta(n) \Rightarrow$ Algorithm B is more efficient than A **asymptotically**.
  - Assume that $T_A(n) = 2n \log n$ and $T_B(n) = 1000n$, than B runs faster than A only if $n > 2^{500}$ - **not practical!**
- **Large leading constants** may occur when algorithms are **excessively complex**.

Fibonacci Sequence

- Leonardo di Pisa (aka *Fibonacci*) was a 13^{th} century mathematician who published work in the area of **population dynamics**

Fibonacci Sequence

- Leonardo di Pisa (aka *Fibonacci*) was a 13^{th} century mathematician who published work in the area of **population dynamics**

- **Example**: Let's say we have two rabbits stranded on a desert isle.

Fibonacci Sequence

- Leonardo di Pisa (aka *Fibonacci*) was a 13^{th} century mathematician who published work in the area of **population dynamics**

- **Example**: Let's say we have two rabbits stranded on a desert isle.

Fibonacci Sequence (2)

Rules:

- We start with a single pair of young rabbits.
- A pair of rabbits cannot breed until they are two months old.
- After they are two months old, they produce another pair each month.
- Rabbits never die.

Fibonacci Sequence (3)

Month 1

Month 2

Month 3

Month 4

Month 5

Fibonacci Sequence (3)

Month 1

Month 2

Month 3

Month 4

Month 5

Fibonacci Sequence (3)

Month 1

Month 2

Month 3

Month 4

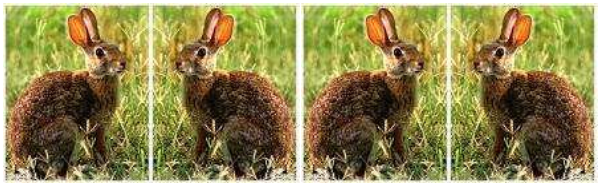
Month 5

Fibonacci Sequence (3)

Month 1

Month 2

Month 3



Month 4

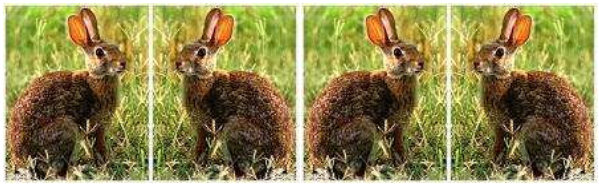
Month 5

Fibonacci Sequence (3)

Month 1

Month 2

Month 3



Month 4

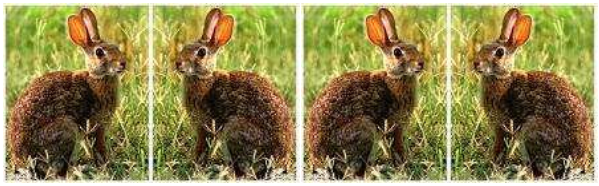
Month 5

Fibonacci Sequence (3)

Month 1

Month 2

Month 3



Month 4

Month 5

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?
Answer: It's hard to see the answer directly, but we can write a **recursive** definition for the problem.

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?
Answer: It's hard to see the answer directly, but we can write a **recursive** definition for the problem.

→ Let $F(n)$ be the number of pairs of rabbits in month n

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?
Answer: It's hard to see the answer directly, but we can write a **recursive** definition for the problem.

- → Let $F(n)$ be the number of pairs of rabbits in month n
- → $F(1) = 1$ (starting pair) and $F(2) = 1$ (since the rabbits are too young to reproduce

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?
Answer: It's hard to see the answer directly, but we can write a **recursive** definition for the problem.

- → Let $F(n)$ be the number of pairs of rabbits in month n
- → $F(1) = 1$ (starting pair) and $F(2) = 1$ (since the rabbits are too young to reproduce

$$F(n) =$$

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?
Answer: It's hard to see the answer directly, but we can write a **recursive** definition for the problem.

- → Let $F(n)$ be the number of pairs of rabbits in month n
- → $F(1) = 1$ (starting pair) and $F(2) = 1$ (since the rabbits are too young to reproduce

$$F(n) = F(n-1)$$

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?
Answer: It's hard to see the answer directly, but we can write a **recursive** definition for the problem.

- → Let $F(n)$ be the number of pairs of rabbits in month n
- → $F(1) = 1$ (starting pair) and $F(2) = 1$ (since the rabbits are too young to reproduce

$$F(n) = F(n-1)+F(n-2)$$

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?
Answer: It's hard to see the answer directly, but we can write a **recursive** definition for the problem.

- → Let $F(n)$ be the number of pairs of rabbits in month n
- → $F(1) = 1$ (starting pair) and $F(2) = 1$ (since the rabbits are too young to reproduce

$$F(n) = F(n-1) + F(n-2)$$

Since rabbits do not die

Fibonacci Sequence (4)

Question: How many pairs of rabbits are there in the n-th month ?
Answer: It's hard to see the answer directly, but we can write a **recursive** definition for the problem.

- → Let $F(n)$ be the number of pairs of rabbits in month n
- → $F(1) = 1$ (starting pair) and $F(2) = 1$ (since the rabbits are too young to reproduce

$$F(n) = F(n-1) + F(n-2)$$

Since rabbits do not die

Number of pairs which are able to breed

Fibonacci Sequence (5)

Month	Reproducing Pairs	Young Pairs	Total Pairs = $F(n)$
1	0	1	1
2	0	1	1
3	1	1	2
4	1	2	3
5	2	3	5
6	3	5	8
⋮	⋮	⋮	⋮

Question: What is an efficient algorithm to calculate $F(n)$ the number of pairs for any month n ?

Fibonacci Sequence - Algorithms

Algorithm 1 - **Recursive Algorithm** which follows directly from the definition of the problem.

```
1  int fib(int n){
2      if(n <= 2)
3          return(1);
4      else
5          return(fib(n-1)+fib(n-2));
6  }
```

Question: Is this algorithm **efficient**?

McMaster University

Algorithm 1 - Recursive Algorithm

```
1  int fib(int n){
2      if(n <= 2)
3          return(1);
4      else
5          return(fib(n-1)+fib(n-2));
6  }
```

$$T(n) = \overbrace{T(n-1)}^{\text{For } F(n-1)} + \overbrace{T(n-2)}^{\text{For } F(n-2)} + \overbrace{2c}^{\text{For add.+compare}}$$

Run Time Growth Rate:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 2c \\ &= (T(n-2) + T(n-3) + 2c) + T(n-2) + 2c \\ &= 2T(n-2) + T(n-3) + 4c \end{aligned}$$

Run Time Growth Rate:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 2c \\ &= (T(n-2) + T(n-3) + 2c) + T(n-2) + 2c \\ &= 2T(n-2) + T(n-3) + 4c \\ &> 2T(n-2) \end{aligned}$$

Run Time Growth Rate:

$$\begin{aligned} T(n) &= T(n-1)+T(n-2)+2c \\ &= (T(n-2)+T(n-3)+2c)+T(n-2)+2c \\ &= 2T(n-2)+T(n-3)+4c \\ &> 2T(n-2) \\ &> 2(2T(n-2-2)) = 4T(n-4) \end{aligned}$$

Run Time Growth Rate:

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 2c \\
&= (T(n-2) + T(n-3) + 2c) + T(n-2) + 2c \\
&= 2T(n-2) + T(n-3) + 4c \\
&> 2T(n-2) \\
&> 2(2T(n-2-2)) = 4T(n-4) \\
&> 2(4T(n-4-2)) = 8T(n-6) \\
&> 2(8T(n-6-2)) = 16T(n-8) \\
&> \cdots
\end{aligned}
$$

Run Time Growth Rate:

$$\begin{aligned}
T(n) &= T(n-1)+T(n-2)+2c \\
&= (T(n-2)+T(n-3)+2c)+T(n-2)+2c \\
&= 2T(n-2)+T(n-3)+4c \\
&> 2T(n-2) \\
&> 2(2T(n-2-2)) = 4T(n-4) \\
&> 2(4T(n-4-2)) = 8T(n-6) \\
&> 2(8T(n-6-2)) = 16T(n-8) \\
&> \cdots
\end{aligned}$$

- In general, $T(n) > 2^k T(n-2k)$

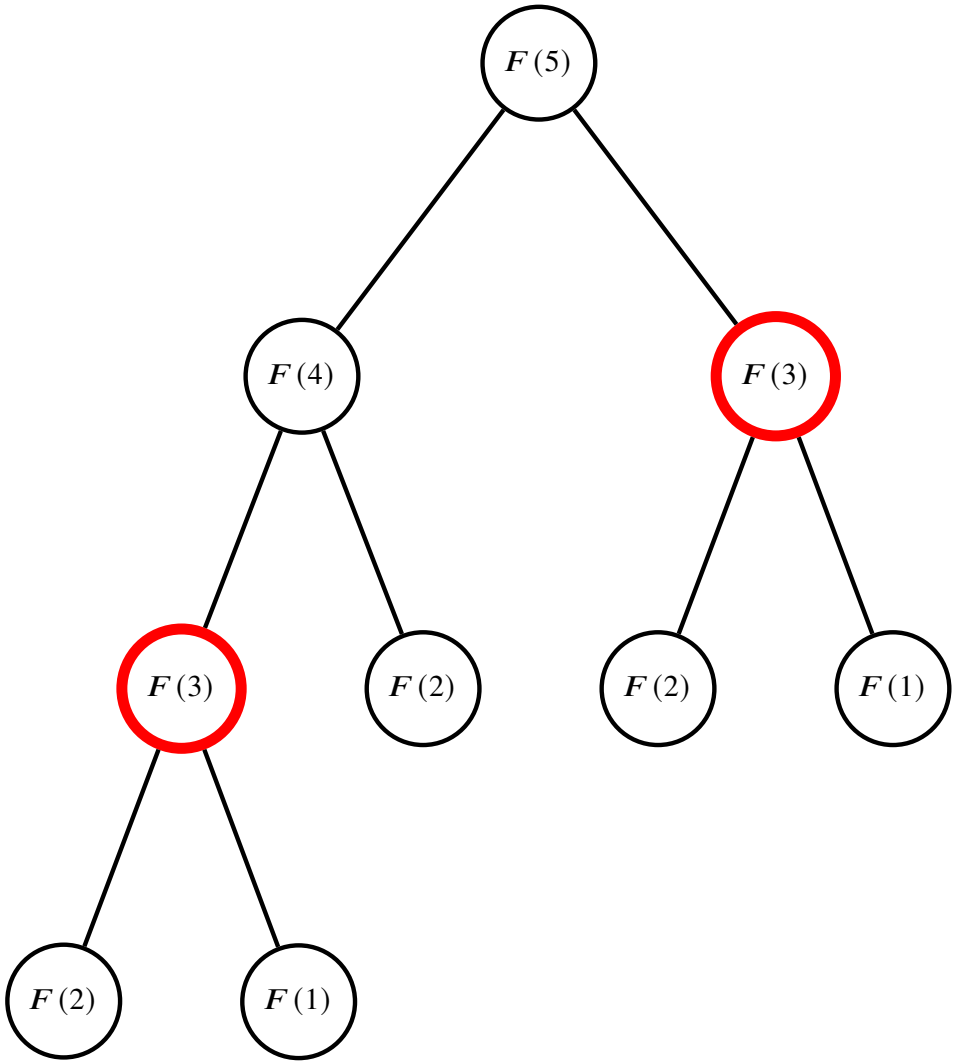
Fibonacci Sequence - Algorithms (4)

- In general, $T(n) > 2^k T(n-2k)$
- Set $k = n/2 - 1$ when n even, or $k = (n-1)/2$ when n odd to make $T(n-2k) = c$.
- Then, for some constant ϵ,

$$T(n) > \epsilon 2^{n/2}$$

i.e., $\boxed{\mathbf{T(n) = \Omega(2^{n/2})}}$ ... **exponential** time complexity!

Fibonacci Sequence - Algorithms (5)



- Recursive Fibonacci algorithm violates **compound interest** rule!
- The same instance of the problem is solved many times. [1]

[1] I. Stojmenovic, *Recursive Algorithms in Computer Science Courses: Fibonacci Numbers and Binomial Coefficients*, IEEE Transactions on Education, vol. 43, no. 3, August 2000.

McMaster University

Algorithm 2 - Iterative Algorithm

```
1  int fib(int n){
2      int curr,prev;
3      if(n <= 2)
4          return(1);
5      curr=prev=1;
6      for(i=3;i <= n; i++){
7          curr = curr+prev;              /* compute F(i) */
8          prev = curr-prev;              /* set prev=F(i-1) */
9      }
10     return(curr);
11 }
```

Runtime:

McMaster University

Algorithm 2 - Iterative Algorithm

```
1  int fib(int n){
2      int curr,prev;
3      if(n <= 2)
4          return(1);
5      curr=prev=1;
6      for(i=3;i <= n; i++){
7          curr = curr+prev;          /* compute F(i) */
8          prev = curr-prev;          /* set prev=F(i-1) */
9      }
10     return(curr);
11 }
```

Runtime: $T(n) = \Theta(n)$

Fibonacci Sequence - Algorithms (7)

- Can we do any better?

Fibonacci Sequence - Algorithms (7)

- Can we do any better?
- → **YES!** There is a **closed form** for $F(n)$ (sometimes referred to as *Binet's Fibonacci Number Formula*):

$$F(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

Run time:

Fibonacci Sequence - Algorithms (7)

- Can we do any better?
- → **YES!** There is a **closed form** for $F(n)$ (sometimes referred to as *Binet's Fibonacci Number Formula*):

$$F(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

Run time: $T(n) = \Theta(1)$

Sensor Subset Selection

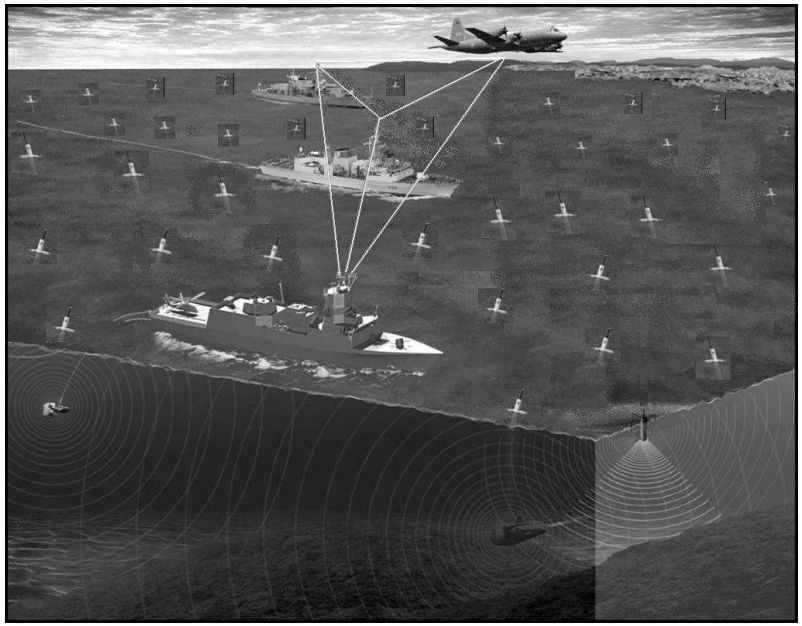


Figure: Anti-submarine warfare

- Sonobuoys are deployed all over the surveillance region from aircrafts.
- Used to detect and track submarines.
- A large number of sensors, N, are available.
- At any one time only a small subset of sensors, n, can be used.

Sensor Subset Selection: Local Search

- Problem: out of N (say 100) sonobuoys, only n (say 10) sonobuys can be used at any one time.
  - Already n sensors are selected by a suboptimal algorithm.
  - Need to find a better sensor subset, which would give better estimation accuracies, by a local search algorithm.
- Local search pseudocode: swap only one sensor at a time

```
1  selectedSensors; //int[n]
2  unselectedSensors; //int[N-n]
3  for (int i=0; i < n; i++){
4      for (int j=0; j < N-n; j++){
5          temporarySelectedSensors = selectedSensors.clone();
6          temporarySelectedSensors[i] = unselectedSensors[j];
7          if (performance(temporarySelectedSensors) >
8                                          performance(selectedSensors))
9          {
10             int swapSensor = selectedSensors[i];
11             selectedSensors[i] = unselectedSensors[j];
12             unselectedSensors[j] = swapSensor;
13         }
14     }
15 }
```

Sensor Subset Selection: Local Search (2)

- Computational time $T(n) = cn(N - n)$
  - $N >> n$
- If n is constant, say 10, the run-time is $\Theta(N)$
- If N and n are variable, the run-time is $\Theta(Nn)$

In the words of Clifford A. Shaffer,

McMaster University

First tune the algorithm, then tune the code.

"The greatest time and space improvements come from a better data structure or algorithm"

Next Topic ...

- Stacks
- Queues