

## 4. Average Time and Probabilistic Programs [WiP]

---

José Proença

Algorithms (CC4010) 2023/2024

CISTER – U.Porto, Porto, Portugal

<https://cister-labs.github.io/alg2324>



**CISTER** - Research Centre in  
Real-Time & Embedded  
Computing Systems

- Measuring precisely performance of algorithms
- Measuring asymptotically performance of algorithms
- Analysing recursive functions
- Measuring **precisely** the **average time** of algorithms
- Possibly: sorting algorithms bubbleSort, swapSort, insertionSort, mergeSort, quickSort
- Next: analysis of sequences of operations (**amortised analysis**)

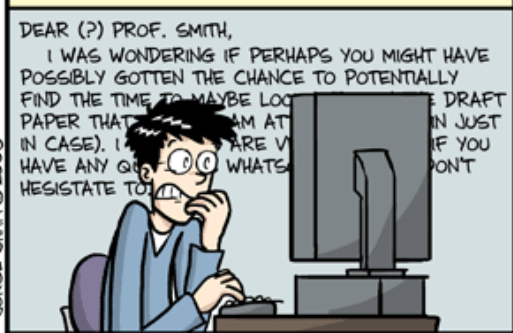
## AVERAGE TIME SPENT COMPOSING ONE E-MAIL

PROFESSORS: 1.3 SECONDS



JORGE CHAM © 2008

GRAD STUDENTS: 1.3 DAYS



WWW.PHDCOMICS.COM

# Recall goal

```
int count = 0;
for (int i=0; i<n; i++)
    if (v[i] == 0) count++
```

## RAM

- worst-case:  $T(n) = 5 + 5n$
- best-case:  $T(n) = 5 + 4n$

## #array-accesses + #count-increments

- worst-case:  $T(n) = 2n$
- best-case:  $T(n) = n$
- average-case:

$$\overline{T}(n) = n + \sum_{0 \leq r < n} P(v[r] = 0)$$

## Preliminaries: series

---

## Recall arithmetic series

$$\sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=a}^b i = a + (a+1) + \dots + b = \frac{(a-b+1)(a+b)}{2}$$

### Intuition

[number of elements]  $\times$  [middle value]

## Recall geometric series I

$$\sum_{i=0}^n x^i = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

### Proof

Let  $S = \sum_{i=0}^n x^i$ . Then:

$$S \times x = x + x^2 + \dots + x^{n+1}$$

Hence we know  $\left[ (S \times x) - S = x^{n+1} - 1 \right]$ .

Simplifying we get  $\left[ S = \frac{x^{n+1} - 1}{x - 1} \right]$ .

## Recall geometric series II

$$\sum_{i=1}^n i \times x^{i-1} = x + (2 \times x^2) + \dots + (n \times x^n) = \frac{n \times x^{n+1} - (n+1) \times x^n + 1}{(x-1)^2}$$

### Proof

Recall  $\left[ S = \sum_{i=1}^n x^i = \frac{x^{n+1}-1}{x-1} \right]$ . Derive both:

$$\begin{aligned} S' &= (1 + x + x^2 + \dots + x^n)' = 0 + 1 + 2x + \dots + n \times x^{n-1} = \sum_{i=1}^n i \times x^{i-1} \\ &= \left( \frac{x^{n+1}-1}{x-1} \right)' = \frac{n \times x^{n+1} - (n+1) \times x^n + 1}{(x-1)^2} \end{aligned}$$



## Calculating average cases

---

The average time to execute an algorithm is given as the **expected value** for its execution, assuming that each run  $r$  has a cost  $c_r$  and a probability  $p_r$ .

### Expected cost

$$\overline{T}(N) = \sum_r p_r \times c_r$$

## Example: Linear search

```
int lsearch(int x, int N, int v[])
{
    // pre: sorted array v
    int i;
    i = 0;
    while ((i < N) && (v[i] < x))
        i++;
    if ((i == N) || (v[i] != x))
        return (-1);
    else return i;
}
```

- Count array accesses
- Best case:  $T(N) = 2$
- Worst case:  $T(N) = N + 1$
- Average case:  $\overline{T}(N) = \dots$

## Example: Linear search

```
int lsearch(int x, int N, int v[])
{
    // pre: sorted array v
    int i;
    i = 0;
    while ((i < N) && (v[i] < x))
        i++;
    if ((i == N) || (v[i] != x))
        return (-1);
    else return i;
}
```

- Count array accesses
- Best case:  $T(N) = 2$
- Worst case:  $T(N) = N + 1$
- Average case:  $\overline{T}(N) = \dots$ 
  - assuming array with uniformly distributed values and a random  $x$
  - same probability to do  $0, 1, \dots, N - 1$  cycle iterations
  - Hence:  $N$  different runs, each
    - probability:  $1/N$
    - cost:  $\#cycles + 1$

## Example: Linear search

```
int lsearch(int x, int N, int v[])
{
    // pre: sorted array v
    int i;
    i = 0;
    while ((i < N) && (v[i] < x))
        i++;
    if ((i == N) || (v[i] != x))
        return (-1);
    else return i;
}
```

$$\overline{T}(N) = \sum_{i=1}^N \frac{1}{N} \times (i + 1)$$

## Example: Linear search

```
int lsearch(int x, int N, int v[])
{
    // pre: sorted array v
    int i;
    i = 0;
    while ((i < N) && (v[i] < x))
        i++;
    if ((i == N) || (v[i] != x))
        return (-1);
    else return i;
}
```

$$\begin{aligned}\overline{T}(N) &= \sum_{i=1}^N \frac{1}{N} \times (i + 1) \\ &= \frac{1}{N} \times \sum_{i=1}^N (i + 1) \\ &= \frac{1}{N} \times \sum_{i=2}^{N+1} i \\ &= \frac{1}{N} \times \frac{N \times (N + 3)}{2} \\ &= \frac{N + 3}{2}\end{aligned}$$

# Binary search

```
int bsearch(int x, int N, int v[])
{
    int i,s,m;
    i=0; s=N-1;
    while (i<s){
        m= (i+s)/2;
        if (v[m] == x) i = s = m;
        else if (v[m] > x) s = m-1;
        else i = m+1;
    }
    if ((i>s) || (v[i] != x))
        return (-1);
    else return i;
}
```

## Ex. 4.1: Calculate best/worst/average cases

- Count array accesses / nr. cycles
- Best case:  $T(N) = ?$
- Worst case:  $T(N) = ?$
- Average case:  $\overline{T}(N) = ?$

## Binary search: Intuition for worst case

- Example:  $N=15$ , worst case
  - 1st cycle: check  $v[N/2]$  (7 remaining)
  - 2nd cycle: check  $v[N/4]$  (or  $v[3N/4]$  – 3 remaining)
  - 3rd cycle: check  $v[N/8]$  (or  $v[3N/8]$ ... – 1 remaining)
  - after: check  $v[N/16]$  (or  $v[3N/16]$ ...) if equal to  $x$
- $N=15$ , (3 cycles)  $\rightarrow$  4 “cycles”
- In general:  $c$  cycles for  $2^c - 1$  elements
- ... i.e.,  $N = 2^c - 1 \quad \equiv \quad c = \log_2(N + 1)$



## Binary search: Intuition for average case

- In an array of size  $N$ , there are  $N+1$  cases (finding at a given position, or not finding).
- Assume  $N+1$  cases have equal probability (!)
- Example:  $N=15$ 
  - 1 cycle: find at  $v[N/2]$  – prob.  $\frac{1}{N+1}$
  - 2 cycles: find at  $v[N/4]$  or  $v[3N/4]$  – prob.  $\frac{2}{N+1}$
  - 3 cycles: find at  $v[N/8]$  or (...) – prob.  $\frac{4}{N+1}$
  - after: find (or not) at  $v[N/16]$  (...) – prob.  $\frac{8}{N+1}$
- $N=15$ , average cycles:  $1 \times \frac{1}{N+1} + 2 \times \frac{2}{N+1} + 3 \times \frac{4}{N+1} + 4 \times \frac{8}{N+1}$
- In general:  $1 \times \frac{1}{N+1} + \dots + \log_2(N+1) \times \frac{2^{\log_2(N+1)-1}}{N+1}$
- ... i.e.,  $\overline{T}(N) = \sum_{i=1}^{\log_2(N+1)} i \times \frac{2^{i-1}}{N+1} = \dots$

# Two's complement

```
void twoComplement(char b[], int N)
{
    int i = N-1;
    while (i>0 && !b[i])
        i--;
    i--;
    while (i >=0) {
        b[i] = !b[i];
        i--;
    }
}
```

## Ex. 4.2: Calculate best/worst/average cases

- Count nr. *bit updates*
- Best case:  $T(N) = ?$
- Worst case:  $T(N) = ?$
- Average case:  $\overline{T}(N) = ?$

# Two's complement

```
void twoComplement(char b[], int N)
{
    int i = N-1;
    while (i>0 && !b[i])
        i--;
    i--;
    while (i >=0) {
        b[i] = !b[i];
        i--;
    }
}
```

## Ex. 4.3: Calculate best/worst/average cases

- Count nr. *bit updates*
- Best case:  $T(N) = ?$
- Worst case:  $T(N) = ?$
- Average case:  $\overline{T}(N) = ?$

twoComplement(0001) = 1111     -   1 vs -1

twoComplement(0010) = 1110     -   2 vs -2

twoComplement(0011) = 1101     -   3 vs -3

twoComplement(01010000) = 10110000

# Exercises

```
int maxgrow(int v[], int N) {  
    int r = 1, i = 0, m;  
    while (i < N-1) {  
        m = grow(v+i, N-i);  
        if (m > r) r = m;  
        i++;  
    }  
    return r;  
}
```

```
int grow(int v[], int N) {  
    int i;  
    for (i=1; i<N; i++)  
        if (v[i] < v[i-1]) break;  
    return i;  
}
```

**Ex. 4.4:** How many comparison of array elements exist in the average case for grow? (assume  $v[i] < v[i-1]$  has 50% chances of succeeding)

**Ex. 4.5:** How many comparison of array elements exist in the average case for maxgrow?

```
void iSort(int v[], int N){  
    int i, j;  
    for (i=1; i<N; i++)  
        for (j=i; j>0 && v[j-1]>v[j];  
            j--)  
            swap(v,j,j-1);  
}
```

**Ex. 4.6:** How many comparison of array elements exist in the average case?  
(as before, assume  $v[j-1]>v[j]$  has 50% chances of succeeding)

# Quicksort analysis

```
int partition(int N, int v[]){  
    int i, j=0;  
    for (i=0; i<N-1; i++)  
        if (v[i]<v[N-1])  
            swap(v,i,j++);  
    swap(v,N-1,j);  
    return j ;  
}
```

```
void quickSort(int N, int v[]){  
    int p;  
    if (N>1) {  
        p = partition(N, v);  
        quickSort(v, p);  
        quickSort(v+p+1, N-p-1);  
    }  
}
```

(See animation at <https://visualgo.net/en/sorting>)

## Partition

- Comparisons:  $T_{\text{partition}}(N) = N - 1$  in any case
- Swaps:  $T_{\text{partition}}(N) = N$  in the worst case, 1 in the best case

## Quicksort (comparisons)

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(p) + T(N - 1 - p) & \text{if } N > 1, \text{ where } 0 \leq p < N \end{cases}$$

# Quicksort – worst case

## Quicksort (comparisons) in general

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(p) + T(N - 1 - p) & \text{if } N > 1, \text{ where } 0 \leq p < N \end{cases}$$

## Quicksort (comparisons) when $p = 0$

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(N - 1) & \text{if } N > 1 \end{cases}$$



# Quicksort – worst case

## Quicksort (comparisons) in general

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(p) + T(N - 1 - p) & \text{if } N > 1, \text{ where } 0 \leq p < N \end{cases}$$

## Quicksort (comparisons) when $p = 0$

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(N - 1) & \text{if } N > 1 \end{cases}$$

$$\begin{aligned} T(N) &= (N - 1) + (N - 2) + \dots + 2 + 1 \\ &= \sum_{i=1}^{N-1} i = \frac{N(N - 1)}{2} = \Theta(N^2) \end{aligned}$$

# Quicksort – best case

## Quicksort (comparisons) in general

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(p) + T(N - 1 - p) & \text{if } N > 1, \text{ where } 0 \leq p < N \end{cases}$$

## Quicksort when $p = \frac{N-1}{2}$

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + 2T(\frac{N-1}{2}) & \text{if } N > 1 \end{cases}$$

# Quicksort – best case

## Quicksort (comparisons) in general

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(p) + T(N - 1 - p) & \text{if } N > 1, \text{ where } 0 \leq p < N \end{cases}$$

## Quicksort when $p = \frac{N-1}{2}$

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + 2T(\frac{N-1}{2}) & \text{if } N > 1 \end{cases}$$

$$\begin{aligned} T(N) &= ???(\text{use recurrence trees}) \\ &= \Theta(N \times \log(N)) \end{aligned}$$

## Quicksort – average case

### Quicksort (comparisons) in general

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(p) + T(N - 1 - p) & \text{if } N > 1, \text{ where } 0 \leq p < N \end{cases}$$

### Quicksort when $p$ can be any with equal probability

$$\overline{T}(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + \sum_{p=0}^{N-1} \frac{1}{N} (\overline{T}(p) + \overline{T}(N - p - 1)) & \text{if } N > 1 \end{cases}$$

# Quicksort – average case

## Quicksort (comparisons) in general

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + T(p) + T(N - 1 - p) & \text{if } N > 1, \text{ where } 0 \leq p < N \end{cases}$$

## Quicksort when $p$ can be any with equal probability

$$\overline{T}(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - 1 + \sum_{p=0}^{N-1} \frac{1}{N} (\overline{T}(p) + \overline{T}(N - p - 1)) & \text{if } N > 1 \end{cases}$$

$$\begin{aligned} \sum_{p=0}^{N-1} \frac{1}{N} (\overline{T}(p) + \overline{T}(N - p - 1)) &= \frac{1}{N} \times \sum_{p=0}^{N-1} \overline{T}(p) + \frac{1}{N} \times \sum_{p=0}^{N-1} \overline{T}(N - p - 1) \\ &= \frac{1}{N} \times \sum_{p=0}^{N-1} \overline{T}(p) + \frac{1}{N} \times \sum_{p=0}^{N-1} \overline{T}(p) = \frac{2}{N} \times \sum_{p=0}^{N-1} \overline{T}(p) \end{aligned}$$

## Quicksort – average case (some math magic)

$$\overline{T}(N) = N - 1 + \sum_{p=0}^{N-1} \frac{1}{N} (\overline{T}(p) + \overline{T}(N - p - 1)) = N - 1 + \frac{2}{N} \times \sum_{p=0}^{N-1} \overline{T}(p)$$

**Multiplying by  $N$**

$$N \times \overline{T}(N) = N \times (N - 1) + 2 \times \sum_{p=0}^{N-1} \overline{T}(p)$$

**Applying for  $N - 1$**

$$(N - 1) \times \overline{T}(N - 1) = (N - 1) \times (N - 2) + 2 \times \sum_{p=0}^{N-2} \overline{T}(p)$$

**Subtracting each side**

$$\begin{aligned} N \times \overline{T}(N) - (N - 1) \times \overline{T}(N - 1) = \\ N \times (N - 1) + 2 \times \sum_{p=0}^{N-1} \overline{T}(p) - (N - 1) \times (N - 2) - 2 \times \sum_{p=0}^{N-2} \overline{T}(p) \end{aligned}$$

# Quicksort – average case (some math magic)

## Subtracting each side

$$\begin{aligned} N \times \overline{T}(N) - (N-1) \times \overline{T}(N-1) &= \\ N \times (N-1) + 2 \times \sum_{p=0}^{N-1} \overline{T}(p) - (N-1) \times (N-2) - 2 \times \sum_{p=0}^{N-2} \overline{T}(p) \end{aligned}$$

## Simplifying

$$\begin{aligned} \overline{T}(N) &= \left( \frac{2N-1}{N} \right) + \left( \frac{N+1}{N} \right) \times \overline{T}(N-1) \\ &= \dots \\ &= \Theta(N \times \log(N)) \end{aligned}$$

# Quicksort – average case (some math magic)

## Subtracting each side

$$\begin{aligned} N \times \overline{T}(N) - (N-1) \times \overline{T}(N-1) &= \\ N \times (N-1) + 2 \times \sum_{p=0}^{N-1} \overline{T}(p) - (N-1) \times (N-2) - 2 \times \sum_{p=0}^{N-2} \overline{T}(p) \end{aligned}$$

## Simplifying

$$\begin{aligned} \overline{T}(N) &= \left(\frac{2N-1}{N}\right) + \left(\frac{N+1}{N}\right) \times \overline{T}(N-1) \\ &= \dots \\ &= \Theta(N \times \log(N)) \end{aligned}$$

**Randomised Quicksort** – the version usually used – uses a **random pivot** when partitioning.



# Randomised Algorithms

---

slides by Pedro Ribeiro, slides 4  
pages 9-13

# Randomized Algorithms

## Randomized algorithms

We call an algorithm **randomized** if its behavior is determined not only by its input but also by values produced by a **random-number generator**

- Most programming environments offer a (deterministic) **pseudorandom-number generator**: it returns numbers that *"look"* statistically random
- We typically refer to the analysis of randomized algorithms by talking about the **expected cost** (ex: the **expected running time**)
- We can use **probabilistic analysis** to analyse randomized algorithms

# Basics of Probabilistic Analysis

- Consider rolling **two dice** and observing the results.
- We call this an **experiment**.
- It has **36 possible outcomes**:  
1-1, 1-2, 1-3, 1-4, 1-5, 1-6, 2-1, 2-2, 2-3, ..., 6-4, 6-5, 6-6
- Each of these outcomes has probability  **$1/36$**  (assuming fair dice)
- What is the probability of the sum of dice being 7?  
**Add** the probabilities of all the outcomes satisfying this condition:  
1-6, 2-5, 3-4, 4-3, 5-2, 6-1 (probability is  **$1/6$** )



# Basics of Probabilistic Analysis

In the language of probability theory, this setting is characterized by a **sample space**  $S$  and a **probability measure**  $p$ .

- **Sample Space** is constituted by all possible outcomes, which are called **elementary events**
- In a **discrete probability distribution** (d.p.d.), the probability measure is a function  $p(e)$  (or  $Pr(e)$ ) over elementary events  $e$  such that:
  - ▶  $p(e) \geq 0$  for all  $e \in S$
  - ▶  $\sum_{e \in S} p(e) = 1$
- An **event** is a subset of the sample space.
- For a d.p.d. the probability of an event is just the **sum** of the probabilities of its elementary events.

# Basics of Probabilistic Analysis

- A **random variable** is a function from elementary events to integers or reals:

Ex: let  $X_1$  be a random variable representing result of first die and  $X_2$  representing the second die.

$X = X_1 + X_2$  would represent the sum of the two

We could now ask: what is the probability that  $X = 7$ ?

- One property of a random variable we care is **expectation**:

## Expectation

For a discrete random variable  $X$  over sample space  $S$ , the expected value of  $X$  is:

$$\mathbf{E}[X] = \sum_{e \in S} \text{Pr}(e) X(e)$$

# Basics of Probabilistic Analysis

- In **words**: the expectation of a random variable  $X$  is just its average value over  $S$ , where each elementary event  $e$  is weighted according to its probability.

Ex: If we roll a single die, the expected value is 3.5  
(all six elementary events have equal probability).

- One possible rewrite of the previous equation, grouping elementary events:

## Expectation (possible rewrite)

$$E[X] = \sum_a Pr(X = a)a$$

# Las Vegas vs. Monte Carlo

- QuickSort always returns a **correct result** (a sorted array) but its **runtime is a random variable** (with  $\mathcal{O}(n \log n)$  in expectation)
- Some randomized algorithms are **not guaranteed to be correct**, but their **runtime is fixed**.

## Las Vegas Algorithms

Randomized algorithms that always output the correct answer, and whose runtimes are random variables.

## Monte Carlo Algorithms

Randomized algorithms that always terminate in a given time bound, but are correct with at least some (high) probability.