

# AK - plan projektu

Porowski Wiktor

8 września 2021

## Spis treści

<b>1</b>	<b>ETAP 1</b>	<b>1</b>
1.1	Wprowadzenie . . . . .	1
1.2	Plan projektu . . . . .	1
1.3	Graf pomocniczy . . . . .	2
<b>2</b>	<b>ETAP 2</b>	<b>3</b>
2.1	Proof of concept . . . . .	3
2.2	Opis testów . . . . .	4
<b>3</b>	<b>ETAP 3</b>	<b>6</b>
3.1	Test . . . . .	6
<b>4</b>	<b>ETAP 4</b>	<b>6</b>
4.1	Omówienie testu i wnioski . . . . .	6

## 1 ETAP 1

### 1.1 Wprowadzenie

Celem projektu jest badanie konsumpcji pamięci RAM przez oprogramowanie tzn. stworzenie kilku wersji algorytmu w mniej lub bardziej optymalny sposób zużywającego zasoby RAM.

### 1.2 Plan projektu

Eksperyment będzie polegał na uruchamianiu różnych wersji algorytmu i obserwacji zużycia RAM przez dany proces. Możliwe będzie

to np. za pomocą narzędzi w Linux.

Następnie porównanie wyników z teoretycznymi przewidywaniami a także badanie rezultatów optymalizacji.

Przydatnym narzędziem może okazać się także debbuger (np. do obserwacji stosu przy wywołaniach rekurencyjnych).

Wybrane problemy to:

1. Obliczanie n-tej liczby ciągu Fibonacciego.

Algorytm ma za zadanie obliczyć wartość liczby na wskazanej pozycji w ciągu Fibonacciego. (skupienie się na iteracyjnej oraz rekurencyjnej wariacji rozwiązania problemu).

2. BFS-przechodzenie grafu w szerz. Dzięki różnym sposobom reprezentacji grafów w pamięci będzie możliwa optymalizacja.

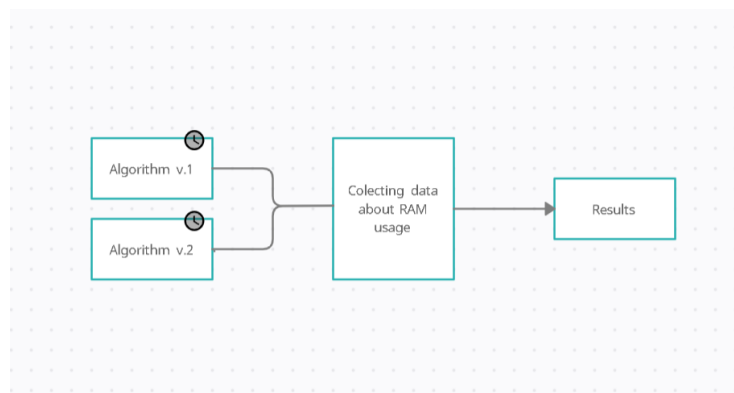
Językiem który zostanie użyty do implementacji algorytmów będzie C++.

System operacyjny - Linux.

Literatura - "Wprowadzenie do algorytmów" Thomas Cormen,

"Architektura komputerów" Janusz Biernat, inne źródła.

### 1.3 Graf pomocniczy



Rysunek 1: Architektura programu

## 2 ETAP 2

## 2.1 Proof of concept

Został napisany kod źródłowy (dołączony w katalogu projektu) oraz zostały dokonane wstępne pomiary zużycia RAM dla różnych wersji algorytmów rozwiązujących dany problem.

Pomiary są wykonywane przez program podczas jego działania i są przekazywane na konsolę.

The screenshot shows the Visual Studio Code interface. The file explorer on the left displays the project structure for 'fibonacci\_recursively.cpp'. The terminal window on the right shows the execution of the program, which prints the Fibonacci sequence from 0 to 15 and the execution time.

```

[Running] cd "/home/akhtar/ak2/project/" && g++ bfs_list.cpp -o bfs_list && "/home/akhtar/ak2/project"/bfs_list
memory at start: 1880
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
memory at end: 1880 + 0
[Done] exited with code: 0 in 0.363 seconds

[Running] cd "/home/akhtar/ak2/project/" && g++ bfs_matrix.cpp -o bfs_matrix && "/home/akhtar/ak2/project"/bfs_matrix
memory at start: 1756
1756
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
memory at end: 1756 + 5172
[Done] exited with code: 0 in 0.455 seconds

```

### Rysunek 2: BFS

[illegible]

Rysunek 3: n-ty wyraz ciągu fibonacciego

## 2.2 Opis testów

Mówiąc o pamięci potrzebnej dla naszego programu mówimy o pamięci potrzebnej na kod, zmienne, dodatkowe alokacje pamięci np (używając malloc) a także współdzielone biblioteki.

Uzasadnione jest aby badając zużycie RAM naszego programu pominąć współdzielone biblioteki ponieważ nie są to zasoby pamięciowe potrzebne specjalnie dla naszego procesu.

Wiele procesów w systemie operacyjnym korzysta z współdzielonych zasobów jak biblioteki.

Tak więc założeniem co do pomiarów jest że chcemy zbadać maksymalną ilość pamięci jaką nasz program może użyć pomijając współdzielone biblioteki. W tym celu możemy skorzystać z getrusage() - zwraca statystyki wykorzystania zasobów dla procesu czyli suma zasobów używanych przez wszystkie wątki w procesie.

Zgodnie z założeniami interesujące dla nas jest ru\_maxrss czyli maksymalny używany rozmiar.



```
DESCRIPTION
getrusage() returns resource usage measures for who, which can be
one of the following:

RUSAGE_SELF
Return resource usage statistics for the calling process,
which is the sum of resources used by all threads in the
process.

RUSAGE_CHILDREN
Return resource usage statistics for all children of the
calling process that have terminated and been waited for.
These statistics will include the resources used by
grandchildren, and further-revealed descendants, if all of
the intervening descendants waited on their terminated
children.

RUSAGE_THREAD (since Linux 2.6.26)
Return resource usage statistics for the calling thread.
The _GNU_SOURCE feature test macro must be defined (before
including any header files) in order to obtain the
definition of this constant from <sys/resource.h>.

The resource usages are returned in the structure pointed to by
rusage, which has the following form:

struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long ru_maxrss; /* maximum resident set size */
    long ru_ixrss; /* integral shared memory size */
    long ru_idrss; /* integral unshared data size */
    long ru_isrss; /* integral unshared stack size */
    long ru_majflt; /* page faults (hard page faults) */
    long ru_minflt; /* page faults (soft page faults) */
    long ru_nswap; /* swaps */
    long ru_inblock; /* block input operations */
    long ru_oublock; /* block output operations */
    long ru_msgsnd; /* IPC messages sent */
    long ru_msgrcv; /* IPC messages received */
    long ru_nsigpend; /* signals received */
    long ru_nvcsw; /* voluntary context switches */
    long ru_nivcsw; /* involuntary context switches */
};

Not all fields are completed; unmentioned fields are set to zero
by the kernel. (The unmentioned fields are provided for
compatibility with other systems, and because they may one day be
supported on Linux.) The fields are interpreted as follows:

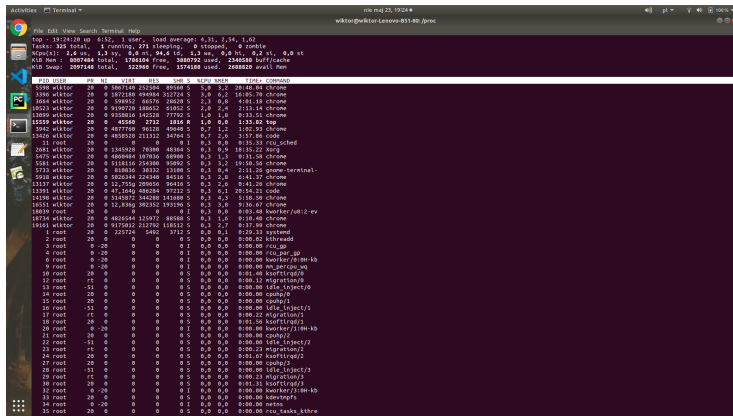
ru_utime
```

Rysunek 4: getrusage()

Praca krokowa w przypadku getrusage():

- Przed rozpoczęciem algorytmu zmierzyć używane zasoby (uwzględnia współdzielone biblioteki)
- Wykonać algorytm
- Zmierzyć zasoby po wykonaniu algorytmu (gdy odejmiemy stan początkowy będziemy mieli informacje o faktycznych zasobach używanych przez nasz algorytm.

W dalszym badaniu RAM używanym przez nasz proces może przydać się pseudoplik /proc/ lub polecenie możliwe jest wydzielenie tak zwanej virtualnej pamięci potrzebnej dla naszego programu (uwzględniając współdzielone zasoby) - VIRT.  
a także ubecznie używane zasoby specyficzne dla naszego procesu - RES.



Rysunek 5: /proc/

Praca krokowa w przypadku tej metody:

- Uruchomić różne wersje algorytmów jako osobne procesy
- Znaleźć ich PID
- Dokonać obserwacji z użyciem /proc/.

## 3 ETAP 3

### 3.1 Test

Plik mp4 z wykonania testów znajduje się w katalogu.

Test 1:

bfs-list 2440 + 0 kb

bfs-matrix 2440 + 4520 kb

fibonacci-iteratively-dynamic 2440 + 0 kb

fibonacci-recursively BRAK DANYCH\*

\*W przypadku elementu ciągu fibonacciego liczonego rekurencyjnie wywołań rekurencyjnych na stosie jest tak dużo że nie jest możliwe otrzymanie wyniku w rozsądnym czasie podczas gdy iteracyjna wersja nie ma z tym problemu.

Test 2:

bfs-list RES-1769 VIRT-14124

bfs-matrix RES-6932 VIRT-17952

fibonacci-iteratively-dynamic RES-1732 VIRT-13988

fibonacci-recursively RES-1868 VIRT-13988

## 4 ETAP 4

### 4.1 Omówienie testu i wnioski

W przypadku elementu ciągu fibonacciego liczonego rekurencyjnie widać że zasoby pamięciowe programu są takie same jak w iteracyjnej wersji.

Uzasadnienie:

W przypadku podejścia iteracyjnego (fibonacci-iteratively-dynamic) ilość wymaganego miejsca pozostaje taka sama. Stąd złożoność pamięci jest  $O(1)$ .

W przypadku rekurencyjnej implementacji fibonacciego wymagana przestrzeń jest proporcjonalna do maksymalnej głębokości drzewa rekurencji, ponieważ jest to maksymalna liczba elementów, które mogą znajdować się w niejawnym stosie wywołań funkcji. Głębo-

kość drzewa rekurencyjnego w tym wypadku jest  $O(n)$ .

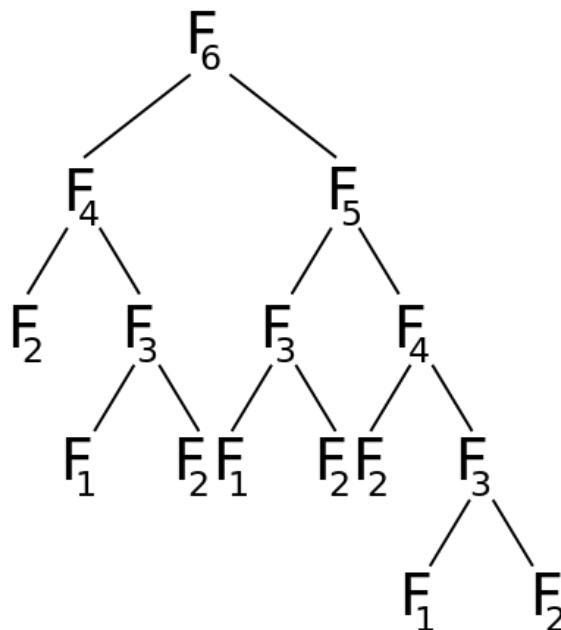
Maksymalny wyraz ciągu który możemy policzyć wykorzystując największy podstawowy typ danych w c++ (long long)

to 92 wyraz = 7 540 113 804 746 346 429 następne wyrazy ciągu nie zmieszczą się w long long.

Tak więc wracając do złożoności pamięciowej  $O(n)$  nasze  $n$  maksymalnie jest równe 92 jest to za mało aby dodatkowa potrzebna pamięć (na stosie) była widoczna w tej metodzie pomiarowej.

Skoro głębokość drzewa to tylko  $O(n)$  dlaczego więc nie otrzymujemy wyniku w rozsądnym czasie?

głębokość drzewa to faktycznie  $O(n)$  jednak liczba wszystkich elementów drzewa - czyli wartości które musimy przetworzyć to  $O(2^n)$



Rysunek 6: rekurencja

Jeżeli chodzi o korzyści z wykorzystania listowej implementacji grafu w stosunku do macierzowej będą one szczególnie wyraźne dla

grafów rzadkich ponieważ implementacja macierzowa posiada złożoność pamięciową  $O(V^2)$  ( $V$  - liczba wierzchołków) natomiast listowa  $O(E)$  ( $E$  - liczba krawędzi). Dla rzadkiego grafu spójnego w przybliżeniu  $E$  równe  $V$  tak więc wtedy  $O(V)$ . Dla grafu bardzo gęstego gdzie  $E$  równe  $V^2$  korzyści się niwelują.